ABSTRACT

Title of Dissertation: MONOLITHICALLY INTEGRATED SRAM-RERAM CACHE-MAIN MEMORY SYSTEM

Candace Walden
Doctor of Philosophy, 2021

Dissertation directed by: Professor Donald Yeung
Department of Electrical
& Computer Engineering

Emerging non-volatile memories are dense and potentially compatible with standard CMOS processes, enabling a monolithically integrated CPU-main memory chip. However, area constraints impact the feasibility of fitting the entirety of a multi-core CPU and main memory system into a single die. ReRAM presents a unique opportunity in that it can be fabricated in crosspoint subarrays which leave the bulk of transistors beneath them available for other logic. However, ReRAM also poses a performance challenge; the latency is generally much higher than that of DRAM. Compensating for this through the increased bandwidth afforded from being on-die poses an architectural problem.

The access circuitry for ReRAM subarrays requires only a small percentage of the area beneath the array. Still, this dense circuitry and wiring disrupts the layouts of irregular logic like CPUs. Caches are very regular and composed of smaller subarrays, making them a better candidate to place beneath crosspoint subarrays.

By co-designing the cache subarrays and ReRAM crosspoint subarrays, minimal disruption to the cache logic can be achieved while still covering the bulk of the last-level cache area in ReRAM.

This work explores the design space when co-designing the last-level cache and ReRAM crosspoint subarrays. Using a modified version of Cacti, we are able to explore the design trade-offs when integrating ReRAM and cache and quantify the impact the ReRAM has on the last-level cache. This design space exploration gives us a first order approximation of the memory capacity of a monolithic computer and informs architectural simulations of such a machine. We also examine how the physical integration presents opportunities for logical integration of the last-level cache and main memory. The interconnects and controllers can be combined, and the addressing can be such that data movement between the main memory and cache is primarily vertical. These optimizations can result in area and energy savings with minor impacts on performance.

The second section of this work explores one architectural style which can balance the monolithic memory system and a general-purpose compute system—a tiled multicore with wide SIMD and multi-threading. We develop a simulator for this architecture capable of simulating a wide variety of system parameters. Through a design space exploration of many of the parameters across sparse, irregular graph kernels and dense, streaming computations, we find monolithic ReRAM exceeds the performance of a state-of-the-art DRAM system for memory intensive workloads given enough parallelism.

We further develop an analytic model to describe our system and highlight

the important performance characteristics for a monolithic CPU-main memory chip. The analytic model is validated against our simulation data. Using this model, we examine the architectural balance of the systems we simulated.

Finally, we develop an RTL model of the combined cache–main memory interface. This gives a more accurate model for the increase in resources required for the combined controller. We additionally develop a system-on-a-chip with an RTL model that alters requests to the FPGA's main memory to be at the speed of ReRAM requests. This model is used to show the performance of more computationally intensive benchmarks. It also is the first step toward creating a test chip for a monolithically integrated ReRAM main memory.

MONOLITHICALLY INTEGRATED SRAM-RERAM CACHE-MAIN MEMORY SYSTEM

by

Candace Walden

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2021

Advisory Committee:
Dr. Donald Yeung, Chair/Advisor
Dr. Bruce Jacob
Dr. Manoj Franklin
Dr. Sahil Shah
Dr. Abhinav Bhatele
Dr. Hanan Samet

To my loving husband, Ben Philip.

Acknowledgments

I would first like to thank my advisor, Dr. Yeung, for his vision, encouragement, and knowledge. His guidance and ability to prompt me to ask the important questions helped me grow immensely.

My thanks goes to all of my committee members for devoting time and energy to my dissertation and defense and providing valuable feedback. In particular, I would like to acknowledge Dr. Jacob and Dr. Peckerar who provided insights and further directions to take my work during our many group meetings.

I also owe a great deal to my lab mates at the University of Maryland, Shang, Meena, Daniel, and Luyi. They offered guidance, levity, and support throughout the process. With a special thanks to Devesh who I closely collaborated with on simulation work and who always offered good recommendations on any problems I encountered.

Finally, I would like to thank my family. My father for instilling a sense of wonder in computers early, my mother for always believing in me, my grandparents for their kindness and interest in me, my siblings for adding some humor and a sense of perspective, and of course, my husband, without whom, this last year would have been impossible.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

As the amount of parallelism in computer systems has increased, so has their demand on memory bandwidth, creating a major challenge for memory technologies. Increasing bandwidth in traditional systems is difficult because all memory traffic must be funneled through a few off-chip channels [5]. This limited connection between where data is needed and where data is stored creates a limit on the bandwidth, often referred to as the bandwidth wall. This can be further exasperated by the trend of large fetch widths to increase bandwidth, as not all the bandwidth is necessarily useful. If a workload is data intensive but has an irregular access pattern, much of the data fetched may be unneeded, decreasing effective bandwidth and increasing power. Additionally, as the amount of data consumed by workloads grows, even useful data movement uses increasingly more power.

To help combat the bandwidth wall and reduce data movement, we propose a monolithic computer. A monolithic computer is one where the CPU and main memory are both located on a single die. This enables extremely parallel connections between the CPU and memory at a fine granularity while minimizing data movement. Processor design has benefited from bringing what were discrete components onto the same die many times—from integrating caches, to memory controllers, to

multiple cores. Just like integrating all the components of a multi-processor on a single die sprung us forward in the 90s, integrating the main memory on the same die should be an important step in the evolution of computers.

1.1 ReRAM and Monolithic Design

The enabling technology for this innovation is emerging non-volatile memories which are compatible with standard CMOS processes and incredibly dense. Whereas DRAM requires special VLSI processes tuned for implementing DRAM's memory cells, some of these new memory technologies can be fabricated today in commercial CMOS fabs at the same technology node as the underlying logic [6]. In addition to some emerging non-volatile memories being compatible with CMOS, researchers have developed memories that allow for 3D stacking of the memory cells to improve density. Examples include Intel's 3D XPoint [7] and Crossbar's 3D ReRAM [6]. In these 3D memory architectures, called "crosspoint architectures", the memory bitcells are sandwiched in between metal wires and individual bitcells are isolated by per-cell "selector devices" rather than access transistors. The use of selector devices enables extremely small bitcells that can be stacked vertically across multiple metal layers. It also means the transistors underneath these sub-arrays are free for implementing unrelated circuits. Some logic is still needed for access circuitry, but the bulk of the transistors are unused.

This thesis focuses on Crossbar's 3D ReRAM. The benefits of this crosspoint ReRAM include higher densities, and thus capacities, than DRAM; lower static

power, as refresh is no longer required; and CMOS compatibility. The crosspoint design and CMOS compatibility implies the ReRAM memory can be fabricated over the CPU, occupying the top-level metal layers of the CPU's die, creating a monolithically integrated CPU–main memory chip. Meanwhile, the CPU's logic can be implemented in the die's logic transistors, minus those needed for the memory access circuits. Putting the CPU and its memory system on the same die will significantly reduce the energy to access memory by reducing data movement considerably, improving power efficiency. It will also allow for an extremely wide connection between the cores and the memory system which will benefit highly parallel architectures.

The drawback of crosspoint ReRAM, though, is that it has a higher latency than DRAM and suffers from a lower endurance. Focusing on architectures that are latency tolerant while bandwidth hungry can help find a niche for ReRAM. Additionally, the entire main memory and CPU must fit on a single die. Integrating the two in a 2D planar fashion leads to the available area becoming a limiting factor, necessitating 3D integration of the memory and CPU logic. This requires that the memory access circuitry be accounted for in CPU layout, possibly creating higher design complexity.

1.2 Contribution 1: Cache–ReRAM Integration

One of the main challenges with a monolithic CPU–main memory die is the area constraints of requiring both systems on a single die. Crossbar's 3D ReRAM allows the CPU to be fabricated beneath the arrays, alleviating the problem to some

degree. Previous work has looked at integrating general CPU logic under crosspoint arrays [8]. They found that placing and routing the CPU required an additional 19% on top of the area requirements of the memory access circuits, with only half the die occupied by ReRAM arrays. This is due to the access circuitry and dense connections to the memory above interrupting routing in the CPU. The irregular structure of the cores is not a good fit for this kind of integration—regular structures like cache do much better. The design we explore integrates the ReRAM over an SRAM last-level cache.

We use Cacti to co-design the ReRAM crosspoint subarrays and SRAM cache mats of the last-level cache (LLC). Using Cacti, we determine the best cache mat size for placement underneath the ReRAM, and then optimize the layout of the integrated cache and ReRAM subarrays to form a complete LLC slice/main memory module. This study shows co-designing the LLC and main memory module saves area with manageable increases in delay and energy for LLC accesses.

1.3 Contribution 2: Tiled CPU-Main Memory Architecture

While the enabling technology for monolithic CPU–main memory dies is promising, it is still rather immature compared to DRAM with some limiting characteristics. Under what conditions the benefits of crosspoint ReRAM arrays outweigh the downsides is a major design question. We have opted to focus on a promising architecture—a tiled CPU that incorporates a large amount parallelism, including thread-level parallelism (TLP) and data parallelism in the form of single instruction

multiple data (SIMD) instructions. This should help hide the latency while providing a large amount of throughput somewhat similar to a GPU but while maintaining a more general-purpose style of programming. Each tile will contain a core, network router, a slice of the last-level cache, a slice of the main memory (potentially decreasing data movement) and a memory controller (increasing memory bandwidth and parallelism).

We also consider the architectural implications of the co-location of main memory and the last-level cache. Their new physical proximity creates an opportunity to streamline the interface between them. By placing the memory system's subarrays above the LLC's subarrays, the internal interconnects between the cache/memory controller and the cache/memory subarrays become redundant and can be reduced to a single interconnect between the controllers and subarrays. This shared interconnect further reduces the area impact of the integration. The cache and memory controllers can also be combined into a single unified controller for both the cache and memory system. Our unified controller retains the cache's MSHRs, but it integrates the memory controller's scheduler queue into them. This streamlining may further reduce area and power requirements with little impact on performance.

1.4 Contribution 3: Simulation

In order to study this monolithic tiled CPU, we have developed a simulator based on our target architecture. The simulator takes an instruction trace from Intel's Software Development Emulator (SDE) [9] and performs a cycle accurate

simulation of the cores, network, cache and memory controllers.

Using this simulator, we have performed many design sweeps for characteristics like the number of banks and latency of ReRAM and cache, the number of threads, the number of tiles and the width of the network. We compare the performance of the monolithic ReRAM system against a state-of-the-art HBM2 DRAM system simulated with our simulator plus DRAMSim3. These simulations show that ReRAM delivers a performance advantage for highly parallel data-intensive computations.

1.5 Contribution 4: Analytic Modeling

The simulator has allowed us to evaluate the monolithic systems performance for particular configurations; however, it is fairly slow, taking days to evaluate a single design point. In order to complete a fuller design space exploration, we have developed an analytic model that captures the important characteristics of the system. This has allowed us to test many more configurations and workload characteristics than the simulator alone.

The analytic model gives us a broad idea of the importance of each characteristic for performance. The model also allows us to look at particular configuration and determine architectural balance for each of the subsystems. We apply this method to the system we modeled in our simulator and show where it is over- or under-provisioned. We look at 14 architectural and workload parameters and show that for most of the workloads we tested, with the baseline configuration, the main memory is still the bottleneck of the system, though the cache is often close to fully

utilized and with reasonable downsizing of the network, it would be as well.

1.6 Contribution 5: RTL Implementation

In order to further explore our monolithic CPU-ReRAM system, we develop an FPGA emulation platform. Open-source cores based on the RISC-V ISA provide a viable platform for customizable system-on-a-chip (SoC) development. An FPGA implementation of our system enables fast performance modeling for benchmarks that benefit from more complex core models than those explored in our simulations. To model the performance of our system, we develop an RTL model that alters requests to the FPGA's main memory to be at the speed of ReRAM requests.

In addition to modeling the ReRAM latency in simulations, we develop an RTL implementation of the combined cache/main memory controller. We use this model to characterize the increased area requirements of our LLC/main memory controller, a component that Cacti does not model.

Beyond providing a fast way to run latency bound benchmarks and characterize the area of the combined cache/main memory controller, this effort will also create many of the resources needed to eventually fabricate a ReRAM test chip.

Chapter 2: Background

2.1 Traditional Memory Systems

Computer memory systems are expected to be affordable, fast, power efficient, and have a large capacity. To accomplish all of these with known technologies, the memory system must be hierarchical. In general, the memory closest to the core is the fastest, least dense, and most expensive. Only a small amount of this memory is available, but it can be accessed without much delay. In further levels, the memory becomes slower but offers greater capacity.

Combining these different memory technologies hierarchically gives the impression of a high speed but large capacity memory. This is due to data locality—if a program recently accessed a memory location it is far more likely to access it or its neighbors soon. These are called temporal or spatial locality, respectively. Storing these memory locations and their neighbors in the fast memory generally results in low-latency accesses. If the entire program and its data does not fit into this fast memory, it can be stored in lower levels and brought into the fast memory when needed, resulting in a large memory capacity.

The traditional levels are the CPU caches composed of SRAM, a DRAM main memory, and finally a permanent storage disk. While the number of caches and the

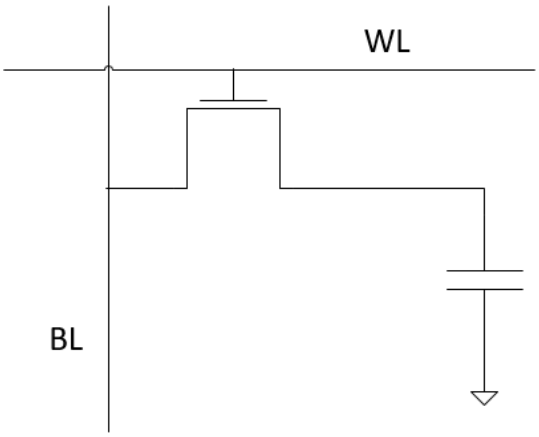


Figure 2.1: DRAM bitcell

nature of the permanent storage has varied over time, these levels in the memory hierarchy have existed for decades.

2.1.1 DRAM DIMMs

The main memory is most often dynamic random access memory (DRAM). DRAM cells are composed of a transistor and capacitor as shown in Figure 2.1. The capacitor holds a charge to indicate the value of the memory. These capacitors are difficult to manufacture on the same die as the CPU, so DRAM is generally on its own die. This style of memory has a relatively low access latency, scaled well, has good write endurance, and is fairly cheap per bit. It has been the choice for computer memories since the mid-1970s. However, as we continued scaling, DRAM has run into challenges with density and power.

One drawback of DRAM is the need for refresh. The value of a bit is stored in a capacitor; this capacitor discharges over time necessitating recharging every so often. As the number of cells has increased, the power required for refresh op-

erations has substantially increased to become a significant component of DRAM energy consumption [10]. Beyond just increasing memory capacity, scaling the capacitor reduces its capacitance, and therefore, the amount of charge it can store. As the DRAM cell has shrunk, retaining the cells' value for the entire refresh period has become increasingly challenging [11]. Furthermore, aggressive scaling means a reduction in reliability of the cells. This is due to interference between neighboring cells, the smaller sensing margin between '0' and '1', and an increase in manufacturing defects at smaller feature sizes [12].

In addition to scaling challenges, DRAM also faces what is termed the "bandwidth wall." DRAM is traditionally a commodity component, sold as dual in-line memory modules (DIMMs) that slot into a mother board and communicate via a standardized protocol determined by the Joint Electron Device Engineering Council (JEDEC). The DRAM DIMMs are not on the same die as the CPU; therefore, the CPU must communicate with them across the motherboard. This creates a bandwidth limit as there is only so many pins the CPU has, limiting the number of connections that can be made to the off-chip memory. As the number of cores in a CPU has increased, this limitation noticeably impacts performance [5]. Increasing cache hit rates, link compression, sectored caches with finer granularity fetches are techniques that can decrease the demand for bandwidth and make the best use of the bandwidth available. However, the fundamental problem of the limited connection between the CPU and main memory remains.

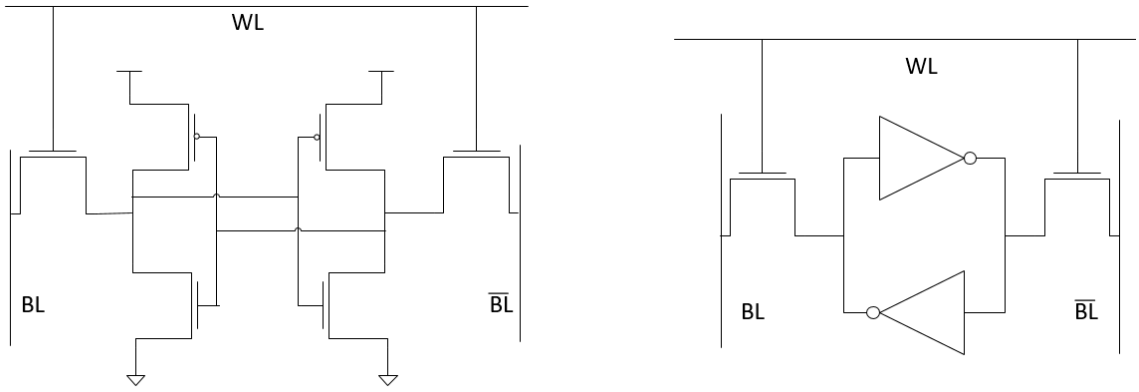


Figure 2.2: SRAM bitcell

2.1.2 SRAM Cache

The cache is traditionally implemented with static random access memory (SRAM). The SRAM cells are composed of six transistors; there are two cross-coupled CMOS inverters and two access transistors as shown in Figure 2.2. The inverters form a feedback loop that maintains the value of the cell as long as there is power. These cells are comparatively large but are very fast and easy to implement on CPU logic dies, important qualities for the lowest levels of the memory hierarchy.

As one way to help alleviate the increasing demand for bandwidth, caches have been expanding in capacity and number of levels. At this point, a level 3 (L3) cache with 10s of megabytes is not uncommon. Due to the large size of the memory cells, this requires substantial area on the CPU die. Four transistor variants of SRAM cells using NMOS logic exist but draw too much power to be used for a large last level cache. Rather than continuing to expand the capacity of the SRAM cache, architects look at making the cache more efficient, such as increasing the hit rate by optimizing the replacement policy [13, 14] and cache compression [15].

2.2 Non-Volatile Memory (NVM)

As the limitations of traditional memory systems are exposed with increasingly parallel and memory hungry workloads, we look to new solutions for our main memory. A promising direction is to use emerging non-volatile memory technology to supplement, or in some cases, even replace, DRAM and SRAM in the memory hierarchy. Examples include resistive RAM (ReRAM), spin-transfer torque magnetic RAM (STT-MRAM), and phase change memory (PCM) [2, 16].

These new memory systems provide much higher capacity at lower cost because the new non-volatile memories are denser and more scalable than DRAM. There is also a significant energy efficiency benefit for these new memory systems, in part because their non-volatility eliminates the need for refresh. One potential drawback of the new non-volatile memories, however, is that they can be slower than DRAM. And writes are more costly than reads, not just in terms of latency, but also in energy and endurance. Addressing these shortcomings has been a major focus of the current research in non-volatile main memory systems.

These non-volatile memories are all relatively new technology, so there is quite a bit of uncertainty as to many of their parameters. This is reflected by the literature where there is often a wide range reported for their characteristics. We give an overview of PCM, STT-MRAM, and ReRAM as well as the expected range for key parameters for each.

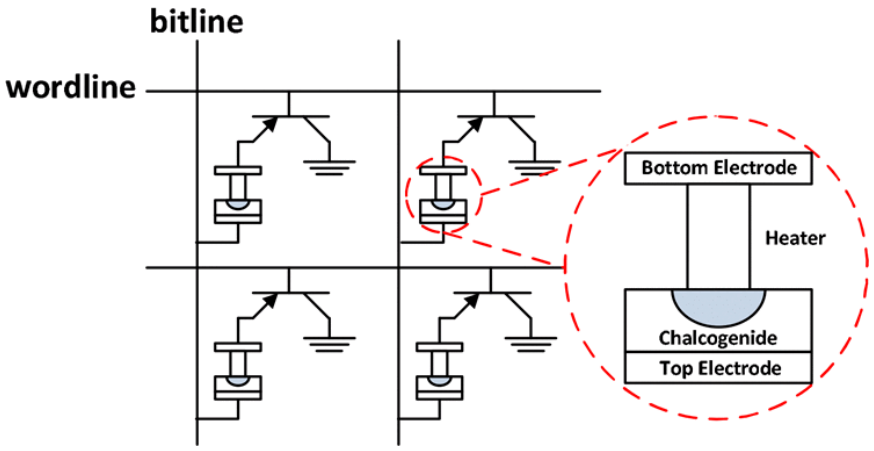


Figure 2.3: PCM bitcells. Source: [2]

2.2.1 Phase Change Memory (PCM)

Phase change memory (PCM or PRAM) is based on changing the phase of chalcogenides from their amorphous phase to their crystalline phase and vice versa [2, 16]. This change in phase is accomplished using heat generated by Joule heating, with current often traveling through higher resistance channels called heaters [17]. Figure 2.3 is an example of this type of memory cell. When in the crystalline phase, the memory cell has a low resistance; when it is in the amorphous phase, it has a high resistance.

Density. The density of PCM is often limited by the access transistor or selectors, as they must be able to pass a large current through the PCM cell to reset the cell [17]. The PCM cell itself can be scaled down to about 1 nm, but the access structure means current cells are about 4-7F^2 [18].

Access Latency. Write latency is generally determined by the set operation as sets must hold the material to a specific temperature long enough for the crystalline structure to form [17]. This set operation can take 10 ns–10 μs depending

on the material and desired retention [18]. Read latency has been cited as 50 ns-100 ns [2, 19], 36 ns [20] and as low as 10 ns [21].

Access Energy. The read energy has been cited as 7–20 pJ/bit [20] and 2 pJ [22]. Write energies have been reported to be 45 pJ/bit [21], 14 pJ/bit for set and 27 pJ for reset [20].

Write Endurance. Write endurances of 10^7–10^{11} have been reported [18] with expectations that the retention could be improved to 10^{15} write cycles with reduced write energy [23].

2.2.2 Spin-Transfer Torque Magnetic RAM (STT-MRAM)

Magnetic RAM (MRAM) is a magnetic tunnel junction with two ferromagnetic layers separated by a thin oxide to act as a tunnel barrier. One of the magnetic layers acts as a reference layer and has a fixed magnetic direction. The other layer is the free layer. If the free layer has the same direction as the reference layer, they are in the parallel state, the MJT resistance is low, and the cell is set. Whereas if they have opposite magnetic directions, they are in the anti-parallel state, the resistance will be high, and the cell is reset. In spin-transfer torque MRAM (STT-MRAM) the magnetic direction of the free layer is changed via spin-transfer torque [2, 16, 24]. An example of an STT-MRAM cell is shown in Figure 2.4.

Density. The density of STT-MRAM is currently 9–51F^2 [2] with 6F^2 and potentially smaller expected in the future as cell design tends toward more 3-D structures such as vertical transistors and multi-level MRAM cells [3].

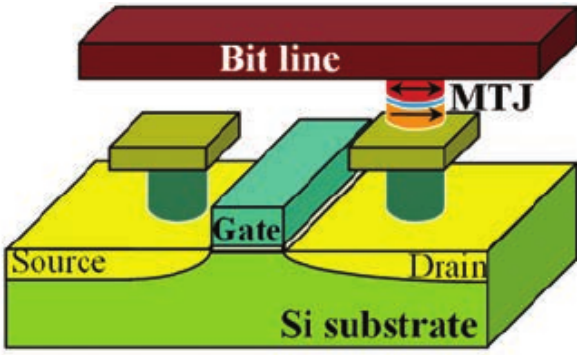


Figure 2.4: STT-MRAM bitcell. Source: [3]

Access Latency. Read latencies have been reported to be 1.4 ns–2.1 ns [2, 24, 25] up to 11 ns [19]. Writes are asymmetric, with setting the cell being faster than resetting the cell [24]. Write latencies have been reported to be 3.9 ns for set and 9.7 ns for reset [24], 6 ns [2], and up to 30 ns [19].

Access Energy. The read energy of STT-MRAM is reported as low as 0.15 pJ/bit [24, 25, 26] to 1.7 pJ/bit [2]. As with most NVMs, writes are more expensive with values reported from 0.25 pJ/bit [24, 25] and 0.6 pJ/bit [26] to 5.8 pJ/bit write [2].

Write Endurance. STT-MRAM does not have issues with endurance unlike most NVMs. Experiments have shown it to have an expected endurance of $> 10^{15}$ write cycles [27], putting it on par with DRAM.

2.2.3 Resistive RAM (ReRAM)

Resistive RAM (ReRAM) cells generally are composed of a metal oxide between two metal layers. A conducting filament can be formed in the metal oxide to connect the two metal layers, putting the cell in a low resistance state, and setting it. Similarly, the filament can be destroyed, putting the cell in a high resistance

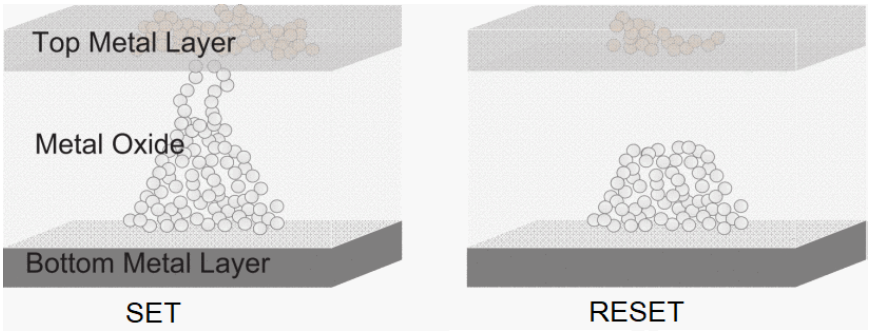


Figure 2.5: ReRAM bitcell. Source: [4]

state, and resetting it [4].

Density. ReRAM cells are minimally sized to the metal layer wires they exist between to give each size an area of $4F^2$ [4]. The cell itself is expected to scale below 10 nm without issue [2].

Access Latency. Read latencies have been reported at below 10 ns [19, 26] on the low end, at 120 ns [28], and 300–600 ns [29] on the higher end. Although there is agreement that writes are more expensive than reads, a similarly wide range of write latencies have been reported. Write latencies from 10 ns [26] and 35 ns [4] to 150 ns [28] and 203 ns [19] or more [29] can be found in the literature.

Access Energy. It is possible that ReRAM will have sub pJ/bit write energies [2] with 1.1 pJ/bit [26] to 105 pJ/bit [19] being reported. Read energies have a smaller range of reported values; values of 0.8 pJ/bit [26] to 1.6 pJ/bit [30] are seen.

Write Endurance. For ReRAM, endurance of 10^6 [28] to 10^{12} [31] have been reported with possible trade-offs between retention and endurance [32, 33].

2.2.4 NVM Systems

Many researchers have proposed memory systems containing such non-volatile memories, either as a replacement for or in addition to traditional memory technologies.

Wu *et al* looked at replacing parts of the SRAM cache with STT-MRAM, PCM and eDRAM to improve capacity and standby power [34]. In one scheme, they replaced last-level cache with an STT-MRAM last-level cache and achieved a 7% performance improvement while reducing power by 53%. In another, they merged the L2 and L3 and had some regions of each be traditional SRAM or one of the alternate memory technologies leading to a 9% gain in performance with similar power improvements.

Researchers have also been looking into using non-volatile memories as the main memory while using a small DRAM cache [35, 36] as visualized in Figure 2.6(a). This allows the large capacity with small power overheads of NVMs while still maintaining most of the DRAM performance, similar to the relationship of SRAM caches and DRAM main memory in traditional memory systems. Since the DRAM is being used as a cache, this memory architecture is transparent to software. This type of system has been shown to result in a 3x speedup [35] or reduce system costs by 40% while maintaining performance [36].

Another scheme to include both DRAM and NVM is a flat hybrid memory system shown in Figure 2.6(b). Some pages of main memory are placed in DRAM and others in NVM. Unlike the DRAM cache architecture, techniques need to be

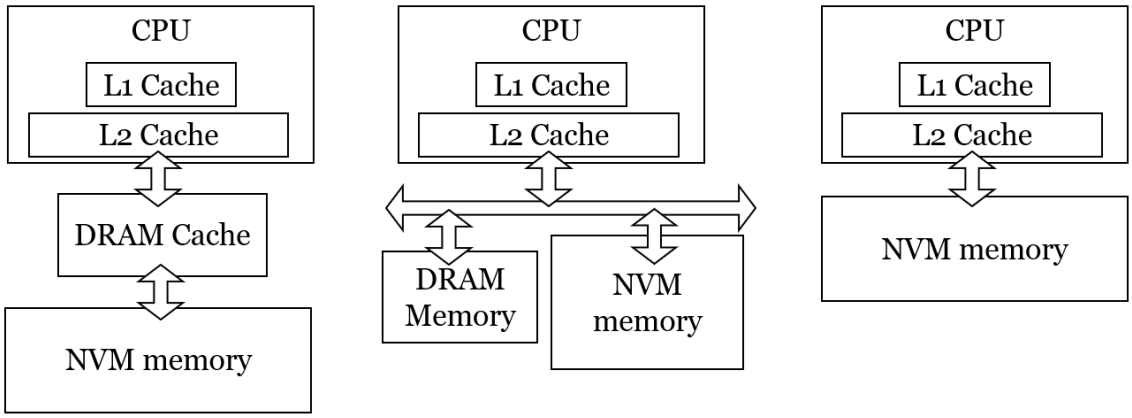


Figure 2.6: Main memory architectures containing NV memory.

developed to determine where to place each page. The memory pages can be placed to minimize write traffic and its associated increased latency, power, and wearout [37, 38], to minimize all traffic to reduce any performance degradation [29, 39], or a combination of the two [40]. The OS can place pages based on hardware counters [37, 38], software analysis at runtime [39] or static profiling [29]; another method is to allow the memory controller to place pages without the OS [40]. All schemes attempt to leverage the benefits of the increased density and decreased static power and cost of NVMs while maintaining the performance of DRAM.

Finally, Xu *et al* studied a ReRAM-based main memory system without accompanying DRAM [4], as shown in Figure 2.6(c). The system uses a compression-based data encoding scheme and a novel write scheduling policy to mitigate the impact of writes and achieve within 10% performance of a DRAM system.

2.3 Single Chip Computer

New memory technologies do not alleviate all the problems of traditional memory systems. They can increase density and reduce power. However, if we place these technologies in DIMMs and access them in the same manner, we will still have the bandwidth limitations and higher communication energy inherent to off-chip channels.

To solve those issues, the memory could be placed on the CPU die to create a single chip computer. This solution has been a goal for decades. However, it comes with its own challenges. Compared to conventional discrete memory systems, this type of design provides less area to fabricate the combined compute and memory circuits. It was once thought that embedded DRAM (eDRAM), which can be integrated into the CPU die, would lead to single chip computers [41]. However, since eDRAM is much less dense than commodity DRAM, it is difficult to achieve the capacity that is required of the main memory in modern systems on the CPU die. However, an additional interesting characteristic of emerging non-volatile memories is their compatibility with CMOS logic while maintaining their density.

Fabricating commodity DRAM requires special VLSI processes tuned for implementing DRAM memory cells. In order to create embedded DRAM, capacitors must be developed in a process which is not specialized to support them [42], reducing density by about 75%–83%. On the other hand, fabricating non-volatile memory can be done within the context of a standard CMOS logic process without any loss of density. This implies we can integrate non-volatile memory directly into the die

of a CPU (or even a GPU) while maintaining density and scalability.

This only goes so far in mitigating the issue of physically placing an entire processor and memory system onto a single die though. If the two are integrated in a 2D planar fashion, then the available area will quickly become a limiting factor. One option is die stacking. This is the most mature way of increasing available area but still suffers from a limitation on the number of connections between the dies. A much less mature possibility is true 3D monolithic integration where layers of transistors can be fabricated on top of each other. However, techniques to reliably manufacture such chips are still being developed. While we cannot yet reasonably fabricate transistors on top of transistors, recent non-volatile memories allow for monolithic 3D stacking of the memory cells in processes that already exist today. This type of integration may be the best option for creating a single chip computer.

2.3.1 Die Stacking

One approach to creating enough die area to implement both the CPU and main memory is stacking memory dies directly on top of the CPU die. In die stacking, two dies are manufactured in a typical 2D logic process. Then these wafers are thinned to 10-100μm in thickness, and thermocompression is used to bond the layers together [43]. This allows vertical stacking of all types of logic, and even mixing and matching different process technologies within a stack [44].

Die stacking memories is a very active, and fairly mature, area of research. Commercial products are available such as High Bandwidth Memory (HBM), de-

veloped by Samsung, AMD and SK Hynix, which is a stack of DRAM memory dies on top of a logic die which contains circuits like I/O buffers and self-test logic [45]. There are also tools being made available to study such architectures in the academic setting, such as DESTINY [46] and 3DCacti [47] which both can model and optimize die stacked caches. Researchers have manufactured test chips with cache over the CPU such as 3D-MAPS [48] and Centip3De [44]; both contain a 64-core processor on their bottom die with a 256 KB SRAM cache on their second die. The Centip3De design includes an additional 64-cores and 256 MB of DRAM in 5 more dies, but the test chip did not.

While many of the stacked die designs contain only DRAM [43, 48, 49], there are an increasing number that include different types of emerging memories for use as either the cache or main memory. Guo *et al* designed a stacked DRAM/PCM memory module that includes both a DRAM cache and a hybrid memory design in a single die stack [50]. Zhang *et al.* analyzed stacking a DRAM/PCM main memory over a core, demonstrating the increased heat generated by the logic will be beneficial to the PCM as less power is needed to perform the thermally driven phase changes [51]. Sun *et al* looked at using MRAM in a stacked die as part of a hybrid cache scheme with a 73.5% reduction in power [52].

Another important area of die stacking research is on improving the connections between the stacked dies, called through silicon vias (TSVs), for particular applications. The shapes and materials of these wires effects how they transmit power [53] and heat [54], and affects overall performance [55]. However, these TSVs still pose a challenge for die stacking. While more plentiful than off-chip pins, there

is a limited number of TSVs since the TSVs are considerably larger than the feature size. Their size is partly due to wafer-to-wafer alignment tolerances during bonding, imposing limitations on their scaling [47]. In addition to limiting the number of connections, their size means the TSVs are generally placed centrally on the chip to prevent disrupting the layout of other logic. This increases data movement and energy and reduces the overall bandwidth and performance of the system.

2.3.2 3D Monolithic Integration

A different solution would be monolithic integration where all of the components of the stack are fabricated on top of each other instead of separately. Traditional processes only allow the active elements, like transistors, to be fabricated in the lowest layers of the stack. The heat required to fabricate conventional transistors would damage the previously created devices if applied later in the process [56]. However, there are emerging nanotechnologies which are compatible with monolithic 3D integration that can act as active devices.

Subhasish Mitra's group proposed monolithic integration enabled by carbon nanotube transistors (CNFETs) for computation, and STT-RAM and ReRAM for cache and memory devices [26, 57]. The bottom layer included conventional CMOS transistors, but there were 3 additional layers containing active devices and several others containing memory cells. The group has demonstrated these devices can be fabricated successfully at 1μm in their nanofabrication facility [58].

Sung Kyu Lim's group focuses on performance optimizations for monolithic

3D ICs [59, 60]. The majority of electronic design automation tools (EDAs) are targeted toward 2D designs, and the ones which do consider 3D are targeted toward die stacking and limited by TSVs. They develop an EDA flow to partition the logic in 3D space with performance as a key consideration [59]. One of the interesting components of one of their designs is placing the memory cells on the bottom of the stack to allow logic cells easy access to the wiring of the memory components [60].

This technology is a very promising direction, but it is still immature with regards toward commercial viability. The components can be fabricated in foundries, but it is still working toward acceptable manufacturing quality, yield, performance, and density for commercial chips [61].

2.3.3 Monolithic Crosspoint Memory

Certain non-volatile memories, such as Intel 3D XPoint [7] and Crossbar 3D ReRAM [6], employ memory cells that are currently being fabricated without per-cell access transistors. In these 3D memories, the bitcells are sandwiched in between metal wires—i.e., at the intersection of wires laid out perpendicularly in adjacent VLSI layers—giving rise to a "crosspoint architecture." Rather than isolate individual bitcells using access transistors, isolation in crosspoint subarrays is provided via "selector devices." This type of monolithic integration can give many of the benefits of 3D monolithic integration without requiring the difficulty of transistors in multiple tiers. In this section, we will focus on Crossbar's ReRAM.

Crossbar's ReRAM bitcells are based on the creation of metallic filaments in

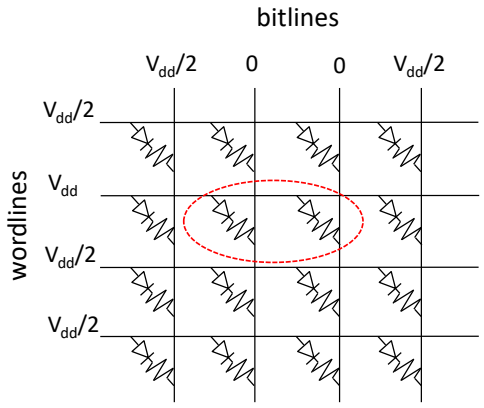


Figure 2.7: Bitcell activation in crosspoint subarrays.

a silicon switching medium [6, 62, 63]. As shown in Figure 2.8, the ReRAM cells lie in between perpendicular wires, in a crosspoint architecture that is fabricated up in the metal stack during back-end of line (BEOL) processing steps. These crosspoint subarrays employ diode-like selector devices integrated in series with the resistive element instead of per-cell access transistors. The selector device within crosspoint subarrays is achieved via a FAST Superlinear Threshold Layer (STL), enabling high selectivity ($> 10^6 - 10^{10}$).

A voltage above a threshold ($> V_{TH}$) must be applied across the selector device and ReRAM bitcell to select the cell and perform a read or write. Figure 2.7 shows the subarray biasing scheme for selection. All wordlines and bitlines of the subarray are held at $V_{DD}/2$, except the selected cell's wordline and bitline are biased to have a difference of V_{DD} across it. The high selectivity of the selector device ensures minimal sneak current on unselected cells, permitting large subarrays (e.g., 2K×2K cells).

Using selector devices instead of access transistors means that multiple layers of such cells can be fabricated to increase density. Figure 2.8 shows two bitcell

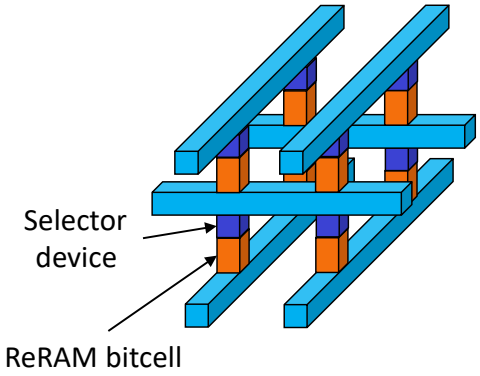


Figure 2.8: Closeup of 3D ReRAM bitcells.

layers in between 3 metal layers. The number of metal layers required to implement n layers of ReRAM is $n+1$. And as many as 8 bitcell layers are possible today. This additional density means capacity can be quite high. Based on Crossbar's experience with their own ReRAM memory chips, 64 GB of ReRAM could be fabricated using 2-stack subarrays and up to 256 GB of ReRAM could be fabricated using 8-stack subarrays, assuming 14 nm on a 400 mm^2 chip [64].

Wordline drive current limitations bound the number of bits that can be sensed along the same wordline to a small number, such as 4 to 8 bits per ReRAM layer. Although cells from different layers can be accessed simultaneously to boost the access parallelism, the biasing scheme in Figure 2.7 prevents adjacent layers from being accessed. (Non-overlapping crosspoints would have to be accessed to utilize different wordlines and bitlines across adjacent layers; however, unwanted selection of off-crosspoint cells would occur in that case). So, only every other layer can be accessed. This means ReRAM crosspoint subarrays inherently exhibit a fine access granularity with two to four byte accesses per 8-stack subarray.

The use of selector devices not only enables the vertical stacking of extremely

small bitcells across multiple metal layers; it also means the transistors underneath the crosspoint subarrays are free for implementing non-memory circuits. This implies the crosspoint memory can be fabricated over the CPU, occupying the top-level metal layers of the CPU's die. Meanwhile, the CPU's logic can be implemented in the die's logic transistors, minus those needed for the memory access circuits. Such placement of the memory system over the CPU can yield higher area efficiency.

Unfortunately, while the memory bitcells do not consume transistors, the non-volatile memory's access circuits (i.e., decoders and sense amplifiers) do. Wordline decoders reside on the side of the array, while column-mux circuits reside on the bottom, forming an "L-shape" as shown in Figure 2.9. The amount of remaining transistor area depends on the size of the subarray, with larger subarrays yielding more unused area. For example, at 14nm and assuming a subarray size of 2K×2K bitcells per layer and 8 layers, we estimate that only 26% of the area underneath each crosspoint subarray would be occupied by the access circuitry, leaving 74% of the area free. Whereas a similar scheme with a subarray size of 512×512 bitcells per layer, only 53% of the area would be free. Instead of leaving this area vacant, these free transistors can be used to implement CPU circuits and achieve higher area efficiency.

When integrated with random logic–e.g., the CPU's datapath–the access circuits can disrupt the compute circuits' layout, introducing significant area overheads [8, 65]. Alternatively, the crosspoint arrays can be integrated over cache. Prior work has observed that the regular structure of SRAM allows it to be placed underneath crosspoint arrays with minimal overheads [8, 32]. In this thesis, we

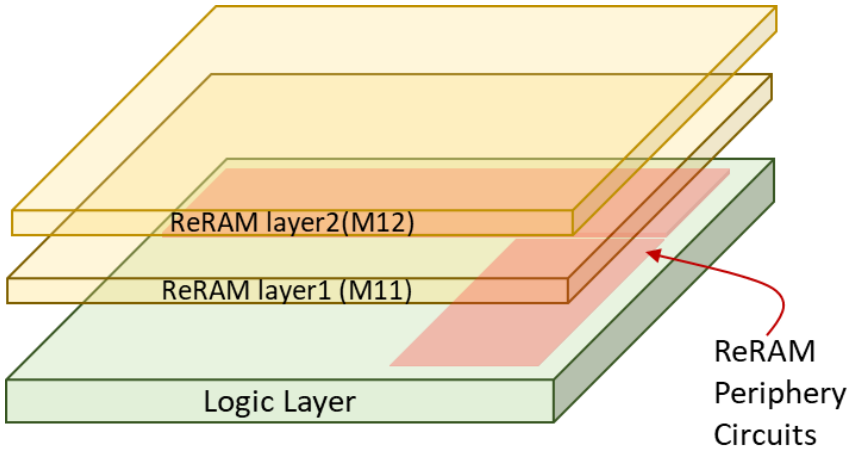


Figure 2.9: The majority of transistors under the subarray is free for non-memory circuits.

exploit this basic observation and present the design of a 3D memory structure comprised of a complete last-level cache (LLC) slice and a crosspoint main memory module.

One concern when placing more components in a smaller area is thermal dissipation and tolerance. The silicon-based switching material used in Crossbar's ReRAM is very stable across a wide temperature range [6], so we do not expect the on-die ReRAM to impose additional heat dissipation constraints. Additionally, in our technique, the ReRAM is not placed above cores but the last-level cache which is generally the coolest part of the CPU. Finally, compared to 3D die stacking, we do not expect heat dissipation to be as problematic for monolithic integration since we only add metal layers on top of a single CPU die. (More heat is trapped between multiple dies in a die stack). Most of the heat will be produced by the logic in the CPU die's substrate which is adjacent to a heat sink.

Another consideration for monolithically integrated main memories is limited

write endurance. Because monolithic integration enables a highly parallel CPU with a high bandwidth memory system, we can expect elevated write frequencies which make write endurance a particularly important issue. For our evaluation in Chapter 5, a write endurance of 10^9 write cycles along with good wear leveling [66] would achieve a system lifetime of 8 years. An endurance of 10^{12}–10^{15} would allow a similar lifetime but without having to use wear leveling. As discussed in Section 2.2.3, ReRAM has a reported endurance of 10^6–10^{12}, suggesting that current ReRAM could possibly be used as monolithic main memory but would need wear leveling. This thesis does not consider endurance techniques. Instead, we assume wear leveling is employed (e.g. [66]) to permit acceptable lifetimes.

Chapter 3: Integrating the Cache and Main Memory

Requiring the CPU and main memory system to fit in the limited area of a single die is a particular challenge of monolithic computers. ReRAM characteristics allow the main memory to be physically integrated over CPU logic, a strategy that may mitigate the issue. However, it is important to consider the integration's impact on the underlying CPU logic.

In previous studies looking at integrating ReRAM over CPU logic, it was found that there is a 19% area penalty from increased routing, with just 48% coverage and unknown impacts on delay and power [65]. Compared to the datapath and control logic previously studied, caches have a very regular layout and minimal connections between internal cache mats. Consequently, crosspoint subarrays can be integrated over cache mats without incurring a routing-induced area penalty. By integrating over the last-level cache (LLC), we can pack the crosspoint subarrays much more densely than over the entire CPU. The LLC is also less sensitive to increases in delay or dynamic power compared to the cores. These characteristics make the LLC a promising candidate for monolithic ReRAM integration.

The only drawback to integrating ReRAM over the LLC, compared to integrating over the cores, is that the amount of ReRAM that can be integrated is

limited by the amount of cache on the CPU die. However, last-level caches can comprise 30-50% of the total die area, so there is ample opportunity for cache-ReRAM integration. For a tiled CPU, each compute tile would own a local slice of the LLC, but the aggregate area of all per-tile slices would still occupy a significant fraction of the die area.

In this study, we focus on the feasibility of tightly integrating ReRAM over an SRAM cache. In doing so, we try to create a general guideline for how much ReRAM can be integrated based on the cache capacity. We also modify the cache design tool, Cacti, to incorporate ReRAM overheads in order to explore how this changes the cache's performance characteristics.

3.1 Cacti

Cacti is a tool maintained by Hewlett Packard to quickly model cache characteristics based on some simple configuration details. It estimates the power, access time, cycle time and area of a cache and can optimize for any of those characteristics. We used Cacti to quickly estimate the feasibility and impact of placing a last-level SRAM cache beneath ReRAM main memory arrays.

Cacti sub-divides caches for performance [67], as shown in Figure 3.1. Each slice of our cache is modeled as its own single uniform cache architecture (UCA) array. An UCA array can be divided into banks which are able to be accessed in parallel. A bank is composed of subbanks; subbanks divide the cache bank bitlines, and only one per bank is accessed at a time. Subbanks are partitioned into mats

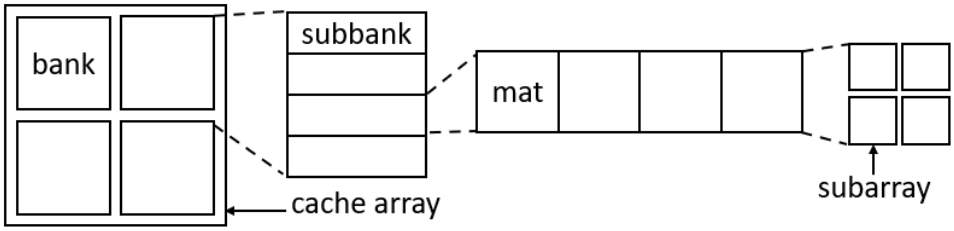


Figure 3.1: Sub-dividing an UCA array.

which divide the wordlines; when a subbank is accessed, every mat is also accessed and provides a fraction of the output. Mats are composed of cache subarrays. The address and data lines thread through the center of the cache mat; other than these limited connections, the cache mat is self-contained.

Cacti allows you to set the number of banks, the number of divisions in the wordline and the number of divisions in the bitline to any power of two. These determine the number of banks, mats and subbanks in the cache array. The number of subarrays per mat is always four.

For an associative cache, the number of ways in a set can also be set to any power of two up to a fully associative cache. The number of sets per wordline can be adjusted: if it is a whole number, multiple sets will be mapped to the same wordline; if it is a fraction, the ways of one set will be mapped to different consecutive wordlines. This parameter, s, allows adjusting the aspect ratio of the mats as seen in Figure 3.2.

We chose the mat as the best cache unit to place beneath the ReRAM arrays. It is the smallest mostly self-contained unit in the cache. The ReRAM arrays are placed with a small gap to allow routing the address and data buses without needing to encounter the dense wiring from the access circuitry to the ReRAM subarrays.

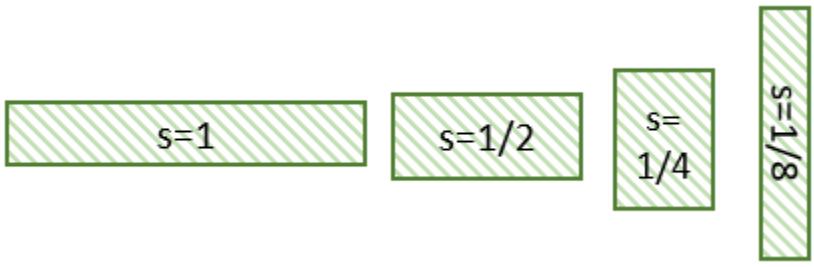


Figure 3.2: s parameter affect on aspect ratio.

We use the parameters to control the number of subbanks, subarrays, and s to find mats which will fit in the empty areas beneath the plausible sizes of ReRAM arrays.

In addition to using Cacti to find the cache mats which will fit beneath the ReRAM banks, we use it to estimate the impact that integrating ReRAM has on the cache. Three or four changes impact the cache characteristics depending on the design. Due to the output width of the ReRAM arrays, it is desirable to have a sectored cache; this introduces additional overheads in the tag arrays. The second is designing the mats to fit in the empty space beneath the arrays. This typically forces the cache to be divided in a less than ideal way. Next, the ReRAM arrays do need to occupy space, which forces the cache arrays to be further apart than they otherwise would. Finally, the ReRAM array may need its own communication interconnect; if this is required, it will add even more area between cache mats. All these of changes impact the expected power and delay of the cache.

Cacti natively calculates the impact of changing the cache mat dimensions, but we needed to modify Cacti to incorporate the new overheads introduced by sectoring and integrating with ReRAM.

3.1.1 Cacti Modifications

One of the major changes we made to Cacti was to the configuration interface. The original Cacti was solely a command line tool, with all parameters being entered as part of the command to run Cacti. Starting in version 6.5, Cacti introduced a configuration file interface. While this is a good step, the file they created does not have a consistent format and is not suitable for scripting runs. We introduced a standard configuration file format (INI) that is supported by scripting languages such as Python.

In addition to standardizing the configuration interface, we added several configuration options to model our ReRAM. We added options to include the ReRAM model for each run, if separate interconnects are needed to communicate with the ReRAM and SRAM, and if an additional network for a directory should be included in the area overhead. To define the ReRAM banks, there is an option for number of ReRAM arrays per mat (either 2 or 4), and the size of each dimension of the ReRAM array. This allows modeling rectangular ReRAM arrays. Additionally, there is a fit goal setting added for optimization use.

Based on these settings, we added the overheads of the ReRAM. The connection from the ReRAM access circuitry to the ReRAM subarrays is very dense. Instead of trying to route through these wires, it is preferable to space the ReRAM subarrays just far enough apart to accommodate any signals that must route through the area. The width of the cache interconnect determines the required gap between ReRAM subarrays as it contains all the signals that must route out of the ReRAM

bank. If separate interconnects are needed, the gap between arrays is double that of the single interconnect. We have also included the ability to add the overhead of a coherence directory interconnect; the directory interconnect would increase the gap by an additional half interconnect width. The ReRAM array area and access circuitry overheads are calculated based on the given size of the array. If the "fit_goal" setting is not set to "DEFINED", the model will select the appropriately sized array based on the mat and goal (under fit, over fit, or best fit). The overheads from the ReRAM arrays were added to the mat model.

Additional configuration options were added to Cacti which are not strictly about modeling the ReRAM. A sectored cache setting was added which increases the number of valid and dirty bits in the tag array from 1 to the number of sectors per tag entry. We also added the option to dictate the layout of banks. Cacti's default behavior is to place the banks in as close to a square configuration as possible, with the long dimension being the horizontal one. The modifications we made allow us to lay the banks out in any rectangular configuration possible. Cacti assumes that the banks themselves will be close to square; the default behavior in those cases is ideal. However, in our designs we often force the banks to be far more rectangular due to how we create the sectored lines; this means we need to be able to control the bank layout to find the optimal configuration. If this option is set to -1, the tool will try all possible bank layouts and select the optimal one based on whatever criteria is set in the configuration file (default is energy-delay square product).

3.2 Area Studies

Using our modified Cacti tool, we can look at the feasibility and impact of tightly integrating the ReRAM and SRAM.

3.2.1 Initial Configuration

Initially, we sought to vary the configuration of the cache to allow the mats to fit beneath ReRAM crosspoint arrays. Cacti limits us to power-of-two subarray sizes for the cache arrays. This restriction means that varying the line size and number of ways per set does not alter the mat area design space as there will always be the same power-of-two capacity cache mats. The mats shown in Figure 3.3 have 64-byte lines and are 8-way set associative. This requirement could be relaxed in practice (for instance having a non-power-of-two number of ways in each set).

The crosspoint subarrays were limited to the 1K $\times$ 1K and 2K $\times$ 2K configurations. These are the smallest, square, power-of-two subarray sizes with enough space to reasonably integrate SRAM arrays beneath. Larger subarrays currently are not advisable, as the number of cells on a single line can significantly increase the sneak current [64]. The number of cells in a crosspoint subarray needs to be limited for optimal power and delay.

The placement of the cache mats under crosspoint subarrays is illustrated in Figure 3.3. Two 2K $\times$ 2K ReRAM subarrays can have a 16 KB cache mat integrated underneath them, four 2K $\times$ 2K ReRAM subarrays fit above a 32 KB cache mat, two 1K $\times$ 1K ReRAM subarrays can accommodate a 2 KB cache mat, and an 8 KB

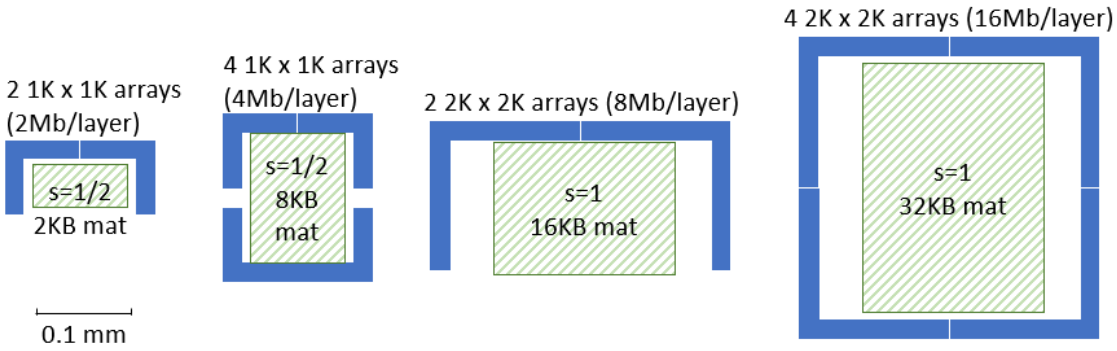


Figure 3.3: ReRAM banks over cache mats

cache mat fits beneath four 1K × 1K ReRAM subarrays. In most of the cases, we were able to place 64x the amount of ReRAM per layer over the SRAM. With up to 8 layers, the main memory system could have 512x the capacity of the last-level cache.

Since we would like to maximize the main memory in this study, we selected cache mats which almost fully fit beneath the crosspoint arrays, minimizing the amount of cache not beneath crosspoint subarrays. If the cache capacity is more critical than the main memory capacity, the fit strategy can be altered. The cache mats could be selected so that they fully occupy the space beneath the crosspoint arrays. This would minimize the amount of empty space beneath the arrays at the expense of some of the SRAM not being beneath the crosspoint subarrays.

The next level of design is that of the cache subbank. The subbank design is primarily based on architectural concerns rather than those rising from integration. Based on our desire to use wide SIMD (AVX-512), we know that cache blocks should be at least 64 bytes to match the maximum fetch size. Additionally, we believe that our ReRAM banks will be able to supply 8 bytes of data per access. If we were

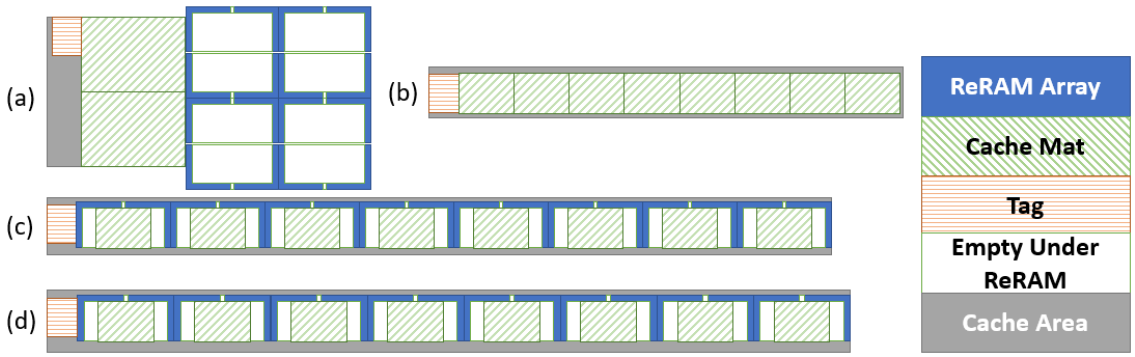


Figure 3.4: 128 KB Cache subbank with 64 Mb of ReRAM/layer. The ReRAM banks are composed of two $2K \times 2K$ arrays. (a) The two not integrated. (b) The cache divided to fit beneath of the ReRAM arrays. (c) The ReRAM integrated over the cache with a single interconnects. (d) The ReRAM integrated over the cache with two interconnects.

to logically integrate the cache and main memory, it would be desirable to have the output of the mats and the ReRAM banks be the same. A 64 bytes line size with mats that each supply 8 bytes requires a subbank composed of 8 mats. This is also the reason for sectoring the cache. Having each bank of ReRAM subarrays be accessed independently from one another is beneficial from a memory parallelism standpoint, which would mean using line sizes of 8 bytes without sectoring. The sectoring allows us to group these lines into a 64-byte line that is more efficient for regular access patterns.

We created subbanks using the mats previously found to fit well with the ReRAM subarrays. We stepped through the various modifications needed to create the final ReRAM configuration and have visualized them in Figures 3.4 and 3.5. For the 2 2K $\times$ 2K subarrays, Figure 3.4(a) shows how the cache and ReRAM are configured before any steps towards integration have taken place; Figure 3.4(b) shows the cache mats being divided to give the proper dimensions; Figure 3.4(c) shows the

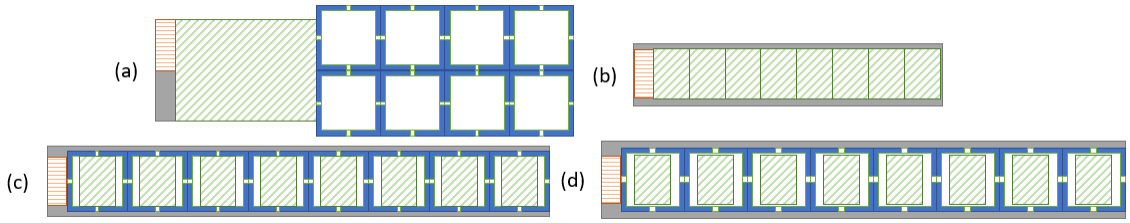


Figure 3.5: 64 KB Cache subbank with 32 Mb of ReRAM/layer. The ReRAM banks are composed of four $1K \times 1K$ arrays. (a) The two not integrated. (b) The cache divided to fit beneath of the ReRAM arrays. (c) The ReRAM integrated over the cache with a single interconnects. (d) The ReRAM integrated over the cache with two interconnects.

ReRAM added to the cache with only a single interconnect; finally Figure 3.4(d) shows the cache with two interconnects—one to communicate with the cache mats and a separate one to communicate with the ReRAM subarrays. Figure 3.5 shows the same steps, in the case of 4 1K $\times$ 1K crosspoint subarrays.

After the design of the cache subbank, we considered an entire cache slice. The cache slice is composed of a number of the previously designed cache subbanks. Once the subbank is determined, there are 3 main design decisions to find the best cache slice—the total capacity of the cache slice, the number of subbanks per bank, and the layout of those banks. The capacity selected will be based on the amount of cache desired and the area constraints of the system. It is not really an optimization problem in this context. The subbanks per bank within a fixed capacity will determine the number of banks per cache slice and therefore the parallelism of the cache. However, in this study, we only tried to optimize for costs like delay and power as measured by Cacti.

Based on our desired design point for the system performance evaluation, we could reasonably have 2 MB caches per tile. This means that for the cache

slice consisting of subbanks designed with ReRAM banks composed of 2 2K $\times$ 2K subarrays there were 16 subbanks, and for the subbanks with ReRAM banks of 4 1K $\times$ 1K subarrays there were 32 subbanks. We swept through all the combinations of subbanks per bank and layout of banks.

There are various trade-offs for area and efficiency vs delay and dynamic energy. Delay and dynamic energy are primarily based on the aspect ratio of the final array. The closer to a square the total array is, the smaller the energy and delay for the cache. The overall area is not as obviously dependent on the layout but has to do with where the bulk of the routing is. Because these are all rectangular areas, some of the area may be unused (e.g., the tag array makes the footprint larger), so these numbers are the worst case area.

Figures 3.6 through 3.11 show the values of the many characteristics we may be trying to optimize for the cache. We would like to minimize the access energy, delay, and area while maximizing the efficiency of our memory arrays (memory array efficiency is defined as the area of memory arrays divided by the total cache area). It is fairly obvious that the very rectangular caches (e.g., 1 $\times$ 32 or 1 $\times$ 16) are poor candidates for integration. Though it is also notable that caches with similar aspect ratios such as 4 $\times$ 4, 2 $\times$ 4, and 1 $\times$ 4 will have different performance especially with regards to area efficiency as seen in Figure 3.11. This demonstrates that the number of subbanks per bank is indeed an important characteristic.

In this study, we wanted to find the effect of integrating ReRAM with the cache and of particular concern the area efficiency of the ReRAM arrays. Thus, for the slice design, we opted to optimize for ReRAM array efficiency. Figure 3.12a

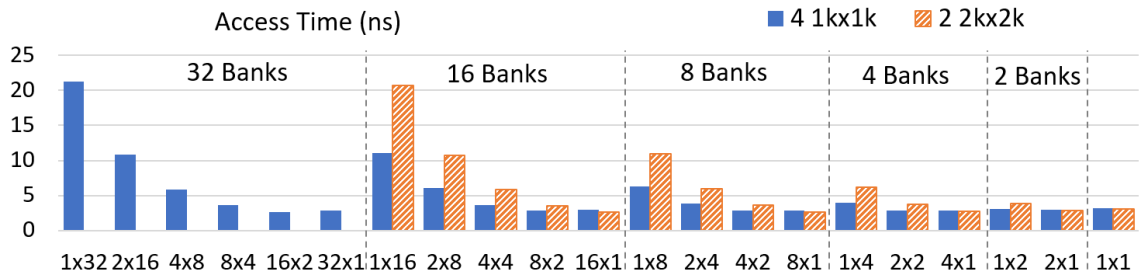


Figure 3.6: Impact of subbanks per bank and bank layout on cache access time.

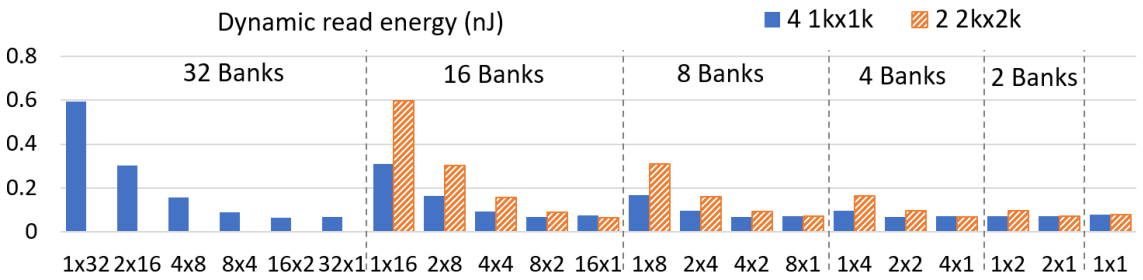


Figure 3.7: Impact of subbanks per bank and bank layout on cache access energy.

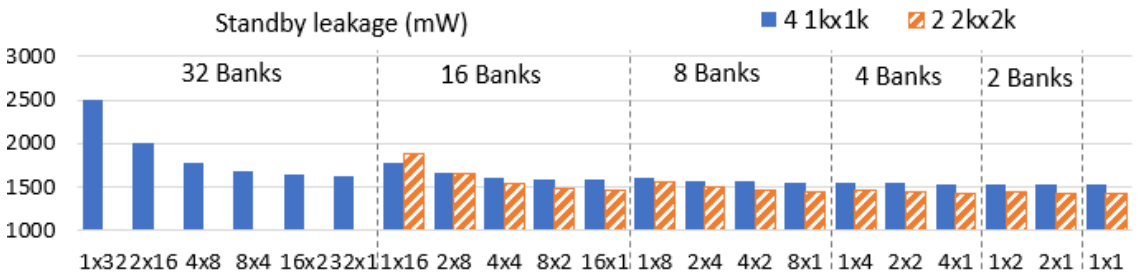


Figure 3.8: Impact of subbanks per bank and bank layout on cache standby leakage.

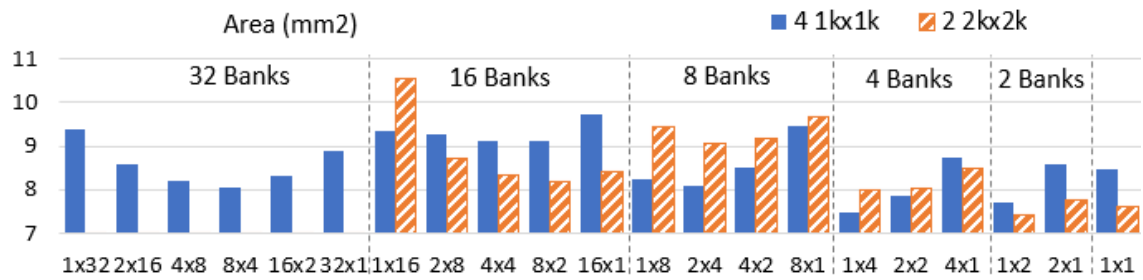


Figure 3.9: Impact of subbanks per bank and bank layout on cache area.

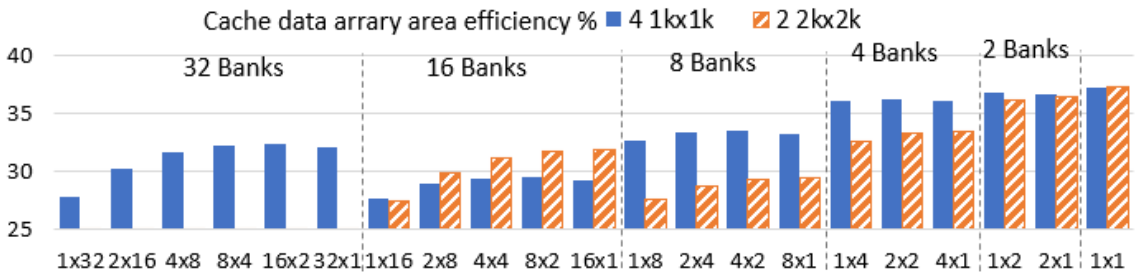


Figure 3.10: Impact of subbanks per bank and bank layout on cache efficiency.

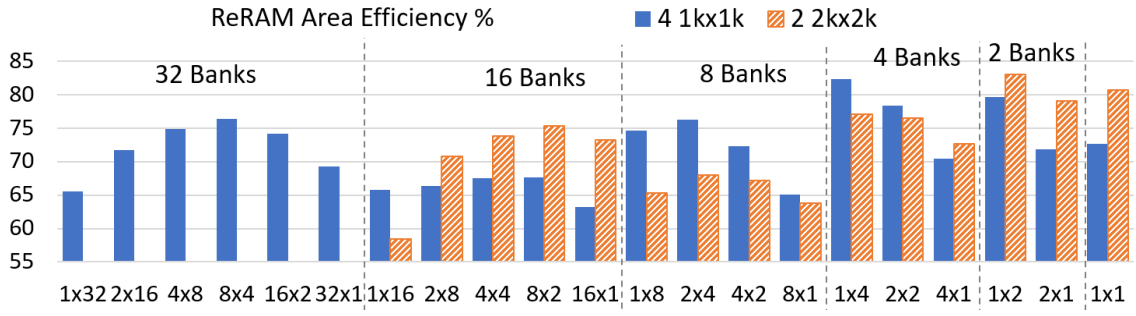


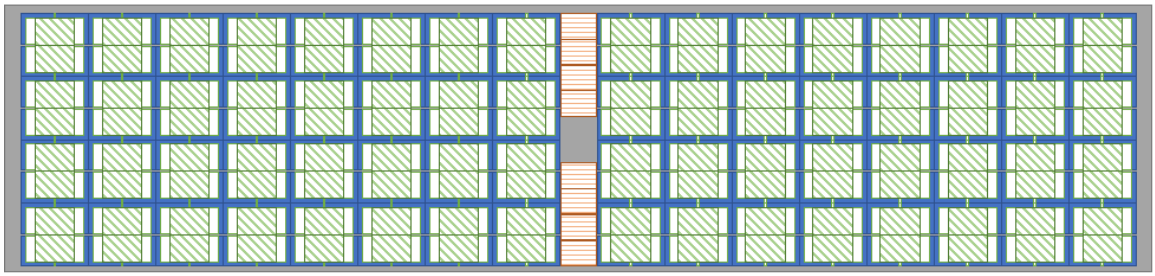
Figure 3.11: Impact of subbanks per bank and bank layout on ReRAM array efficiency.

shows the best cache slice with regards to ReRAM array efficiency for the 2 2K × 2K subarray ReRAM banks. Figure 3.12b shows the same amount of ReRAM and cache if the ReRAM was not integrated with the cache.

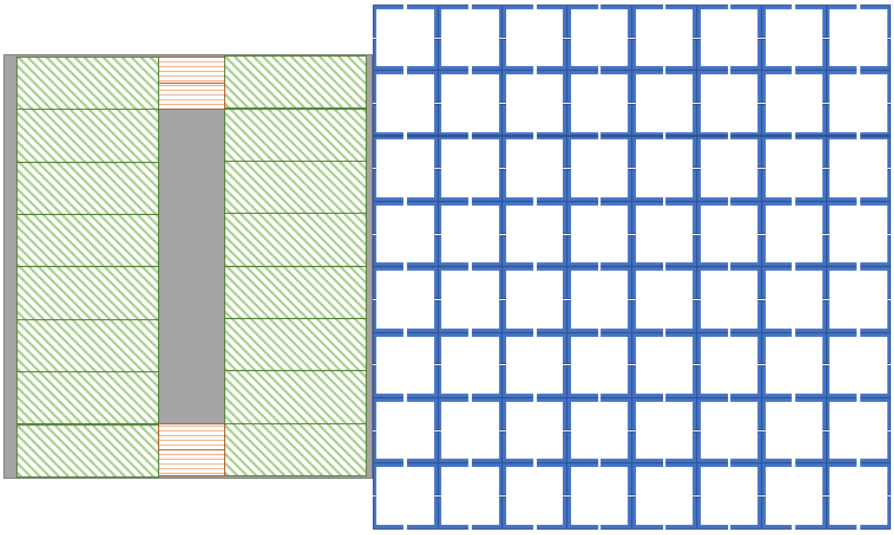
3.2.2 Rectangular ReRAM Arrays

One way to expand our design space is the inclusion of rectangular ReRAM arrays. We look at 4k×1k and 2k×1k arrays. Again, we have a limitation of about 1–4 Mb arrays due to increasing capacitance as we add memory cells and a desire to have enough space beneath the arrays to integrate a meaningful amount of SRAM.

Figure 3.13 shows the integration of 1k×2k and 2k×1k arrays over cache mats, while Figure 3.14 shows the integration of 1k×4k and 4k×1k arrays over cache mats. Both have the same ReRAM to SRAM ratio as most of the previous integrations,



(a) A 2 MB L2 slice organized as 2 cache banks in a 1x2 grid when co-designed with 128 8 MB ReRAM banks.



(b) The same amount of cache and main memory designed separately.

Figure 3.12: Comparison of co-designed to non-integrated 2 MB cache tile. (Drawn to scale).

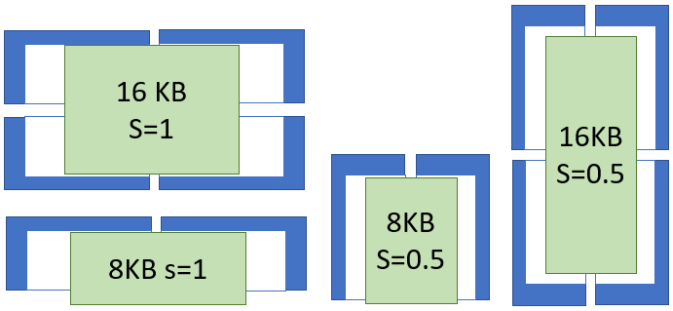


Figure 3.13: 2k×1k and 1k×2k ReRAM arrays over cache mats.

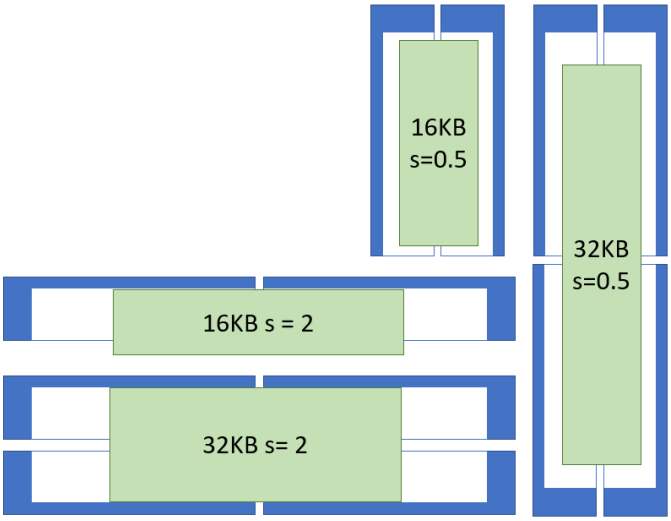


Figure 3.14: 4k×1k and 1k×4k ReRAM arrays over cache mats.

with 2 KB of cache to 1 Mb/layer of ReRAM. The arrays have about the same unused area as previously, but the changed aspect ratios result in different cache performance.

After determining the cache mat–ReRAM bank pairs, we developed the subbank based on the same architectural considerations as the previous section. This means there are 64 byte lines with 8 mats per subbank that provide 8-byte data fetches per mat to output match the ReRAM banks. These subbanks serve as building blocks for larger caches.

We swept all combinations of subbanks per bank and layout of banks for the

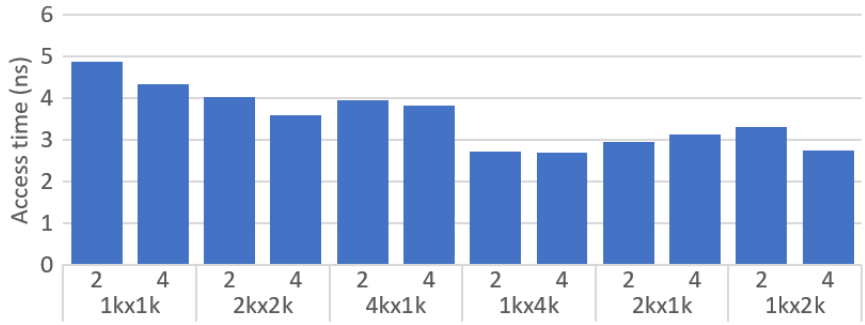


Figure 3.15: Cache access time for different ReRAM array configurations.

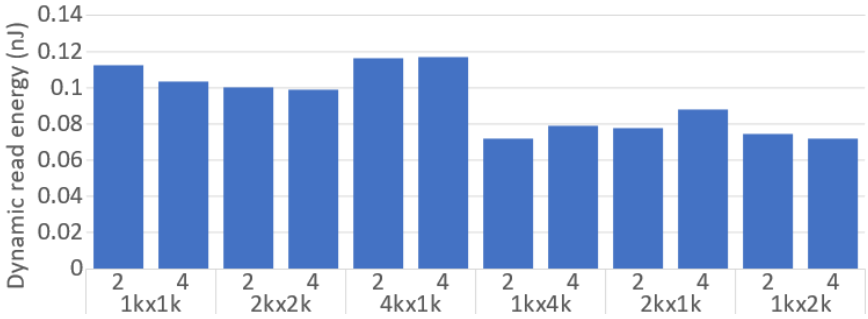


Figure 3.16: Cache dynamic read energy for different ReRAM array configurations.

new ReRAM bank configurations for a 2 MB cache slice. We selected the "best" cache layout for each ReRAM bank configuration. When selecting the "best" cache layout, we prioritized ReRAM area efficiency, but would select a slightly less area efficient layout if the power and delay were significantly better. Figures 3.15–3.19 show the value of the delay, power, area and ReRAM area efficiency of each of the selected caches.

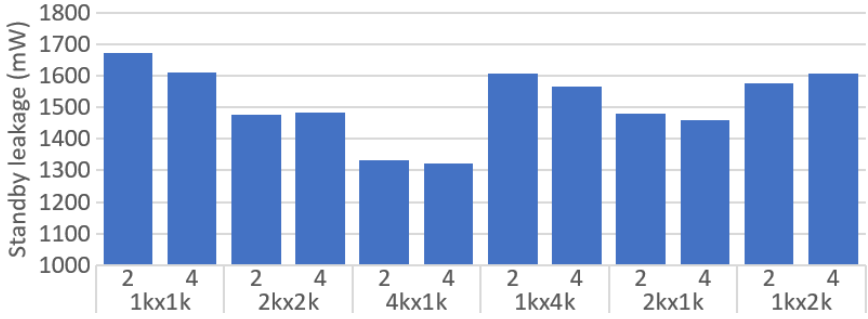


Figure 3.17: Cache standby leakage for different ReRAM array configurations.

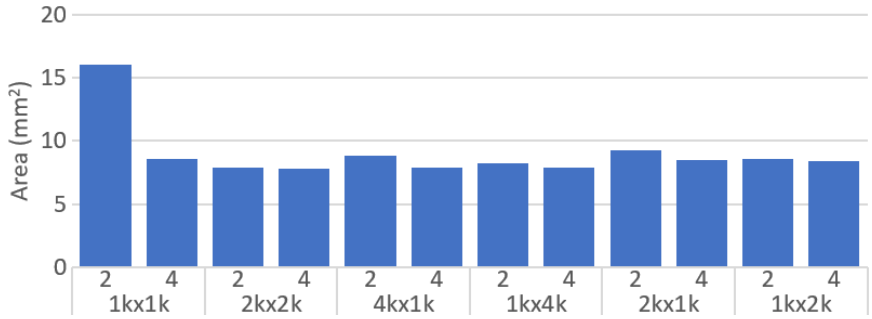


Figure 3.18: Cache area for different ReRAM array configurations.

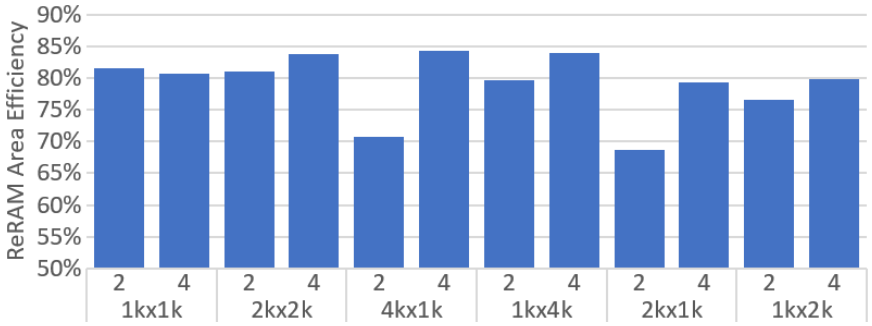


Figure 3.19: ReRAM array area efficiency for different ReRAM array configurations.

We can see that different aspect ratios for the ReRAM arrays and cache mats produce different results, even if their capacities are identical. When comparing four 2k×2k subarrays, four 1k×4k subarrays, and four 4k×1k subarrays per ReRAM bank, they are all placed over a 32 KB cache mat and have a total of eight cache subbanks. However, how those subbanks are organized is different. For the caches with four 2k×2k subarrays and four 1k×4k subarrays per ReRAM banks, the cache is best organized as eight banks composed of a single subbank with eight and four banks per row, respectively. While the four 4k×1k per ReRAM bank is best organized as a single bank with eight subbanks. If we were to organize the other two as a single bank, they would be worse in every cost metric we use but standby leakage. If we were to organize the four 4k×1k per ReRAM bank as eight banks, placing all eight in a single row would be best, and would slightly improve delay and dynamic

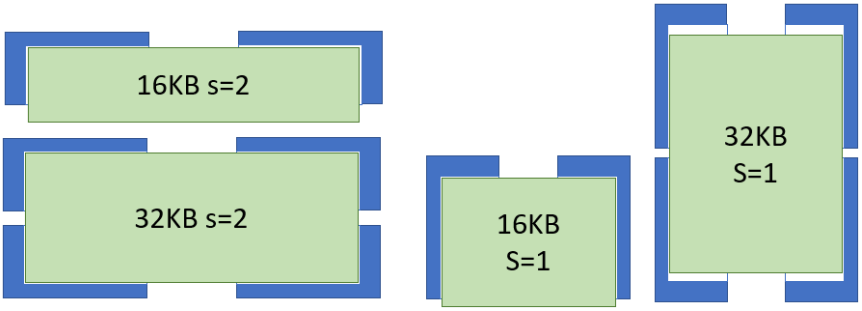


Figure 3.20: 2k×1k and 1k×2k ReRAM arrays over cache mats with cache capacity prioritized over ReRAM capacity.

power, but at the cost of decreasing ReRAM area efficiency from 84% to 76% and increasing standby leakage.

The different aspect ratios have trade-offs. For similar ReRAM area efficiencies, the cache employing 1k×4k subarrays has better dynamic read energy, access time, and parallelism while the 4k×1k has better standby leakage, with the 2k×2k subarrays often between the two. A similar trend, though not as pronounced, can be seen when using the 1k×2k and 2k×1k ReRAM subarrays. Architects will likely need to select the right aspect ratio and orientation of the ReRAM subarrays to meet their design priorities.

3.2.3 Prioritizing Cache Capacity

While selecting cache mat/ReRAM bank pairs, we have opted to prioritize ReRAM capacity. This means selecting mats which will (almost) fully fit beneath ReRAM arrays to maximize the area the ReRAM occupies. We could instead prioritize cache capacity and select mats which (almost) fully occupy the area beneath ReRAM arrays. Such configurations are visualized in Figure 3.20.

When comparing Figure 3.20 to Figure 3.13, we see the capacity of the cache mat has doubled and the number of sets per line, s, also doubled to maintain the aspect ratio. For the same amount of cache capacity, this results in a 50% reduction in ReRAM capacity, but a 36% area decrease. If we compared to an integrated main memory/cache slice with the same ReRAM capacity, the SRAM capacity is doubled with only a 26% increase in area. For caches with the same capacity, there will be a smaller impact on all the cache costs as we are integrating less ReRAM and interrupting the cache design less.

When using the Cacti tool to automatically select ReRAM arrays, this type of fit is classified as *UNDERFIT* because the ReRAM arrays are undersized compared to the cache mats.

3.2.4 Integration Impact

Although integrating ReRAM over cache does not cause routing-induced area penalties like it does when integrating ReRAM over random logic, there are nevertheless penalties to the cache. This is due in part to the cache mat size required to fit beneath crosspoint subarrays being smaller than the optimal mat size. The expansion of the cache to incorporate the ReRAM circuitry can also negatively impact the cache performance.

The penalties to the cache's access latency, dynamic read energy, and standby leakage for each step of integration is presented in Figures 3.21–3.23. Each bar represents one of the integration steps (a visual representation of these steps can

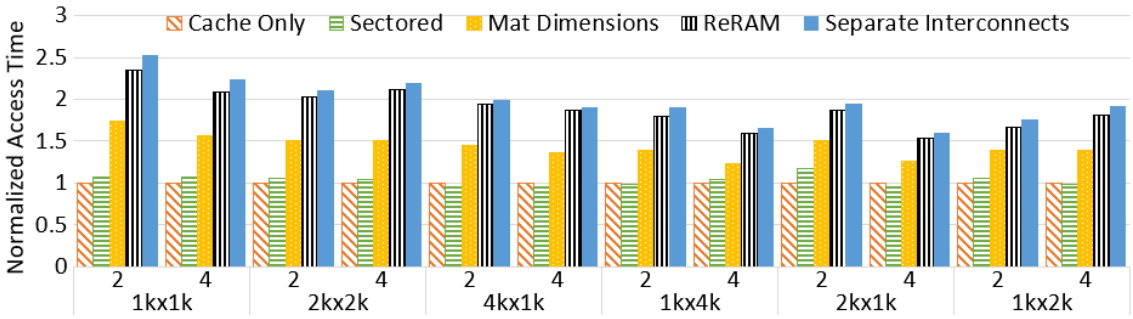


Figure 3.21: Impact of the ReRAM integration on the cache delay.

be found in Figure 3.4). Each value is normalized to the case where the ReRAM and cache are not integrated, and Cacti is free to choose how to organize the array with the selected number of cache banks ("Cache Only"). The first integration step we look at is the impact of sectoring the cache to allow the 8-byte and 64-byte accesses ("Sectored"). The third bar shows the impact of changing the mat size to fit beneath the crosspoint subarrays ("Mat Dimensions"). The fourth bar shows the impact of actually integrating the arrays ("ReRAM"). The final bar is the impact of requiring separate interconnects to communicate with the cache subarrays and the ReRAM subarrays ("2 Interconnects").

Figure 3.21 shows the relative access times for the different cache configurations. The cache access time, in the best case, is 54% more than that of the non-integrated cache and 89% worse on average. If two interconnects are required, this rises to 60% best case and about double on average. This is not too much of a concern for the last-level cache. The access latency to the last-level cache is generally dominated by the time to reach the cache, not the access time [68]. Since we only intend to integrate ReRAM over a distributed last-level cache, the latency increase should be well tolerated.

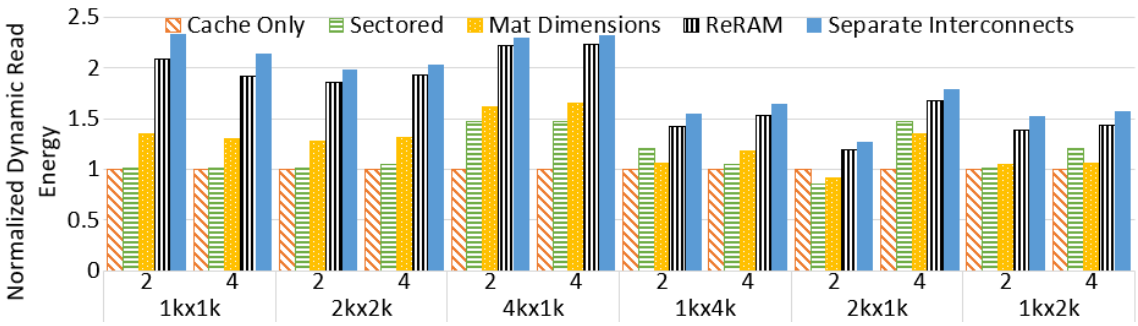


Figure 3.22: Impact of the ReRAM integration on the cache access energy.

The dynamic read energy, presented in Figure 3.22, has a best case increase of 27% for two interconnects, with the average case being 87% greater. If a single interconnect can be used, this overhead reduces to 19% in the best case and 74% on average. The dynamic energy is primarily driven by the increased size of the interconnect. Again, due to the nature of the LLC, the increase in power does not create a huge impact. Even with an increase, the cache access energy is very small compared to that of the memory system. Doubling it creates no noticeable difference in dynamic energy based on our simulations discussed in Chapter 5.

The standby leakage power of the cache is the main driver of standby power in the memory system as the ReRAM has very low static power and does not require refresh. In the integrated cache, it increases primarily due to the increase in access circuitry as we increase the number of mats per subbank and in total. The relative values for the standby leakage are shown in Figure 3.23. In the best case, it increased by 11% and on average it increased by 22% for a single interconnect and 24% for two interconnects.

Figure 3.24 shows the area savings from tightly integrating the cache and ReRAM. The smallest cache mat-ReRAM subbank pair (2 1k×1k ReRAM subarrays

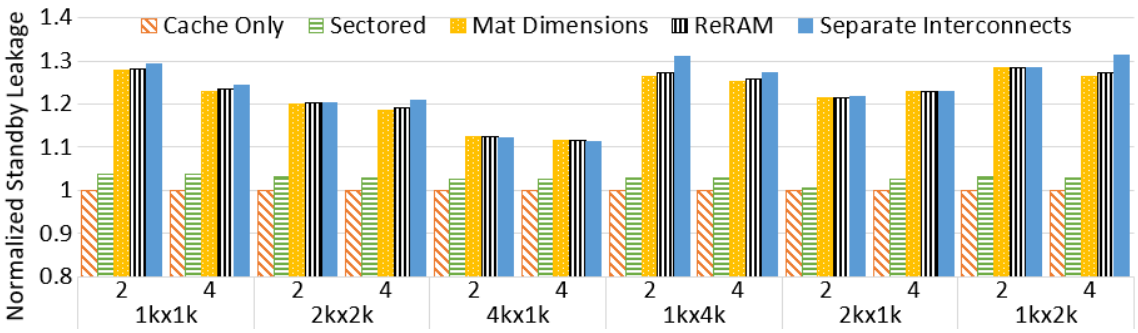


Figure 3.23: Impact of the ReRAM integration on the cache standby leakage.

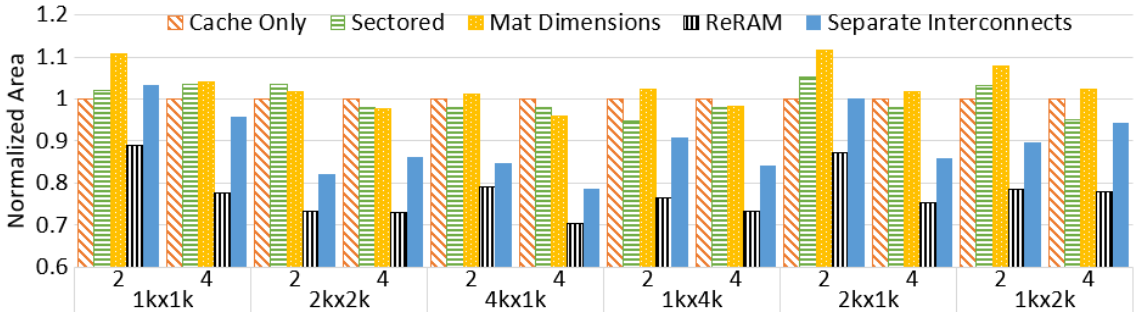


Figure 3.24: Overall area of the cache and ReRAM at different levels of integration, normalized to the "No Integration" case.

over a 2 KB cache mat) fared poorly, only decreasing the area by 11% when fully integrated and increasing the area by 3% if separate interconnects are required. The additional access circuitry and interconnect routing required per mat is a much larger portion of the area than the SRAM subarray compared to other cases. The larger mat-subbank pairs had the greatest area savings, with a 21% decrease in area in the case of independent interconnects and a 30% decrease if a shared interconnect could be used for the 4 1k×4k ReRAM subarrays over the 32 KB cache mat. The average area savings was 22% for a single interconnect and 10% for two interconnects.

Figure 3.25 shows the ReRAM coverage for the different configurations. In the best case, 84% of the cache is covered by crosspoint subarrays with acceptable impacts to the last-level cache. At 28nm, assuming half a 4cm^2 CPU die is dedicated

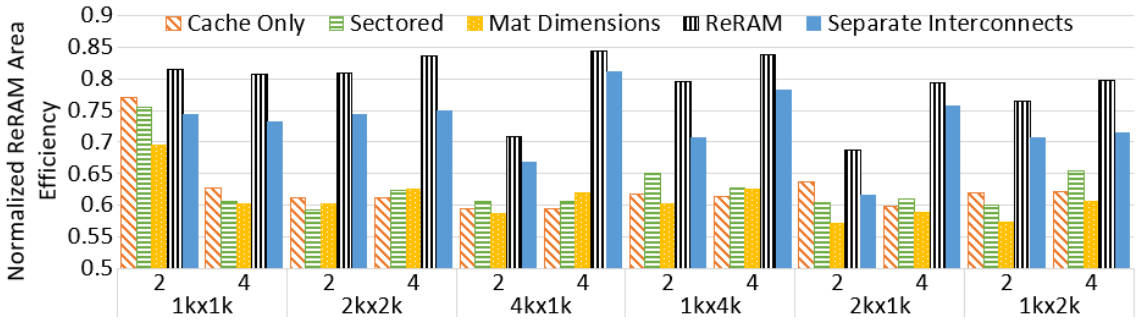


Figure 3.25: Percentage of the area covered by ReRAM arrays for the different configurations.

to ReRAM and cache, 36GB of ReRAM can be integrated with 72MB of cache. ReRAM is expected to scale to at least 7nm. At these more advanced technology nodes, 100s of GBs of ReRAM over 100s of MBs of cache would be possible.

3.2.5 Capacities

With scaling and different possible tile configurations, each slice of the last level cache may have a different capacity than the 2 MB we have been assuming. To see the impact of capacity on the integrated design, we applied all previous subbank configurations for 256 KB to 8 MB cache slices, sweeping through all subbank and bank layouts for the capacity.

This sweep of all configurations results in many different cache designs. Selecting the "best" configuration from these depends on the design goals. We selected all the Pareto optimal design points with respect to delay, dynamic read energy, standby leakage, and ReRAM array area efficiency. In Figure 3.26, we have plotted the delay, dynamic read energy and standby leakage against the ReRAM area efficiency of each of the Pareto optimal points for each cache capacity. Designers will

need to determine the priority of each of these costs and how to balance them in their particular system.

Often the configurations with the best standby leakage have a high delay and dynamic energy; three of these configurations are particularly obvious for the 1 MB cache. Generally, prioritizing ReRAM efficiency results in higher cache costs, but there are many cases where reducing the efficiency a few percent from the maximum gives much more favorable delay and power parameters. This is highlighted in the larger cache sizes which have clusters in the upper right that quickly drop to a more typical value when moving even a small bit toward less area efficient ReRAM configurations.

Increasing the size of the cache results in configurations that can give a better ReRAM area efficiency. However, larger caches also require more power, area and have a higher delay. This isn't a clear-cut trade-off though. When choosing many smaller cache slices over fewer large cache slices, much of the dynamic power and delay will be shifted to the network to reach each of those slices.

To see if the size of the cache has an impact on how much the integration affects the cache, we look specifically at the "best" configuration with four 2k×2k

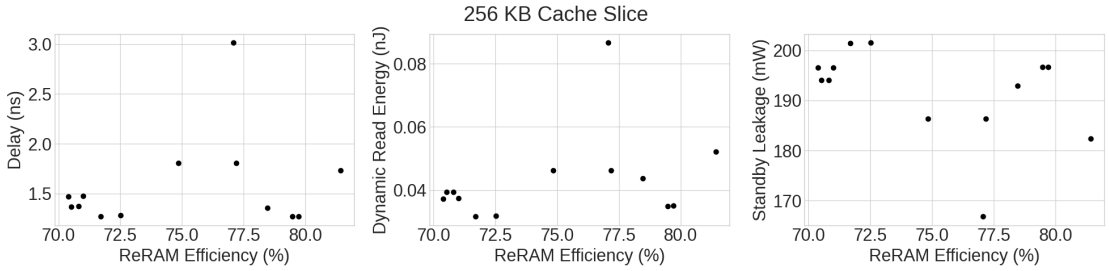


Figure 3.26: Pareto optimal design points for different cache slice capacities.

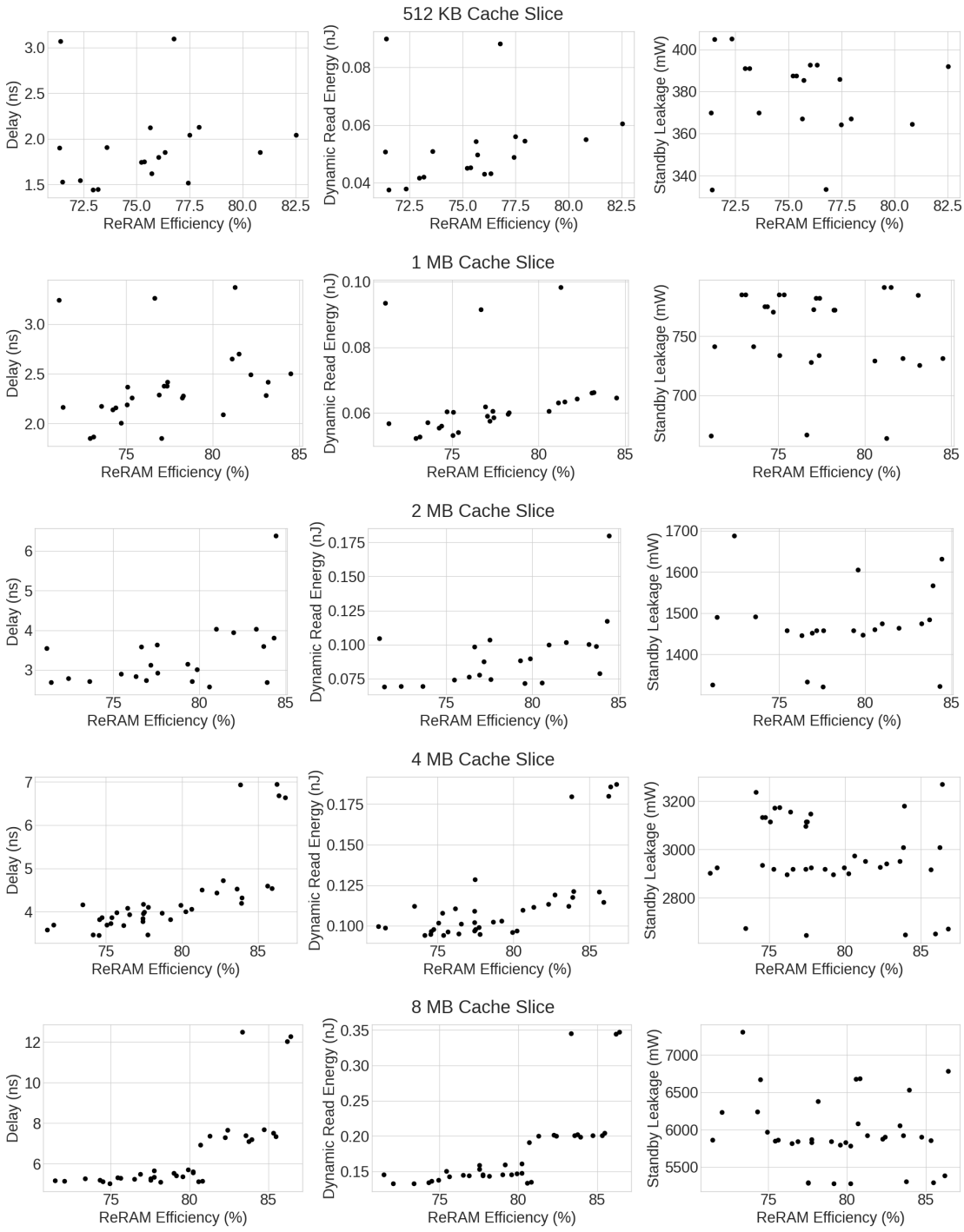


Figure 3.26: Pareto optimal design points for different cache slice capacities.

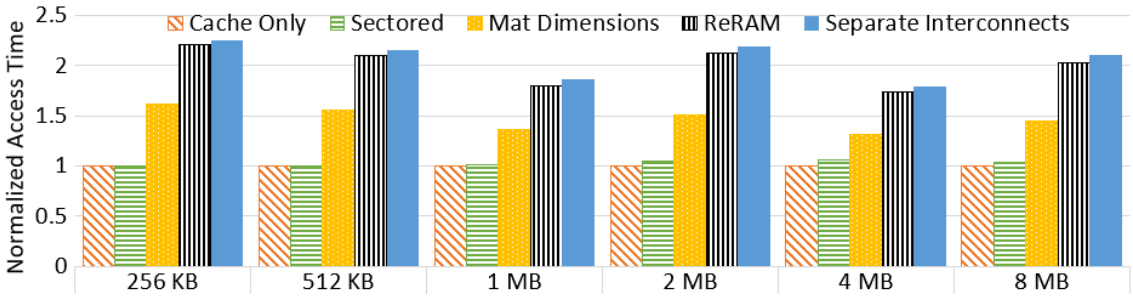


Figure 3.27: Impact of capacity and ReRAM integration on the cache delay.

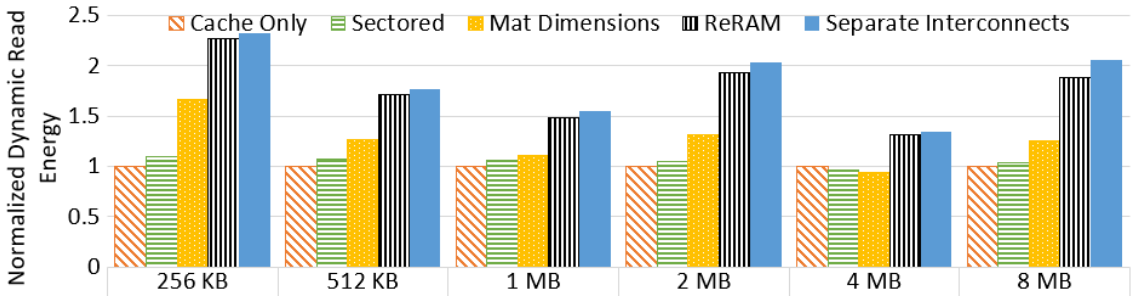


Figure 3.28: Impact of capacity and ReRAM integration on the cache access energy.

ReRAM subarrays per cache mat for each cache capacity. While selecting the "best" configuration per capacity is not a clear-cut problem, we used similar criteria as previously: prioritize ReRAM area efficiency but sacrifice small amounts of it for significantly better power or delay. Figures 3.27–3.29 show the integration impact for each cache capacity with the "best" cache with four 2k×2k ReRAM subarrays per cache mat.

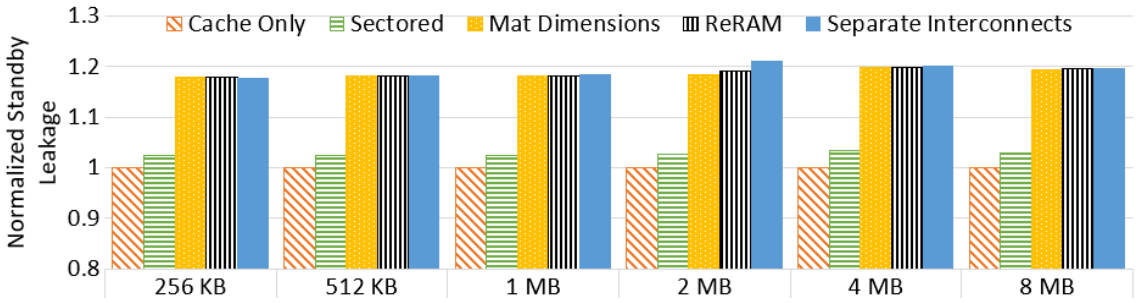


Figure 3.29: Impact of capacity and ReRAM integration on the cache standby leakage.

The only consistent trend we can see in the integration impacts is that larger caches generally have a greater increase in standby leakage compared to smaller caches. The baseline configuration for the larger caches starts with larger mat sizes. For the 256 KB cache, the cache mats initially have a capacity of 32 KB whereas for the 8 MB cache, the mats are initially 128 KB. This means the larger caches have a greater change to the number of mats in order to fit them beneath the ReRAM arrays, resulting in a larger impact to standby leakage. Otherwise, the integration impact seems to be much more dependent on the individual configuration than the capacity of the cache.

3.3 Conclusion

We have demonstrated the feasibility of integrating an SRAM last-level cache with a crosspoint ReRAM main memory. In doing so, we have further developed Cacti as a tool to allow developers to design their own integrated SRAM-ReRAM blocks and evaluate the trade-offs for their specific architecture. In general, with 8-layer ReRAM, we can integrate 1 GB of ReRAM over 2 MB of SRAM cache.

When evaluating the best configuration for a 2 MB cache slice, we found that monolithic integration of the cache and main memory saves up to 30% of area and covers 84% of the cache slice with ReRAM with tolerable increases to delay and power. Depending on the size of the last level cache, this can result in hundreds of gigabytes of main memory on a single die.

For larger cache slices, the impact of integration does not change for delay

or dynamic power but increases slightly for static power. This is one of the many trade-offs designers can evaluate when selecting their cache design with the help of our modified Cacti tool.

Chapter 4: Tiled CPU-Main Memory Architecture

4.1 Tiled Manycore Architecture

Finding an architecture which would benefit from ReRAM's higher parallelism while not being handicapped by the higher latency that it currently exhibits is a challenge for ReRAM. General-purpose multicore CPUs with a few ILP cores are unlikely to benefit from on-die ReRAM. One possible architecture is a many-core CPU with a large number of in-order multi-threaded cores that can gainfully use the memory-level parallelism of on-die memory systems. We select a few key characteristics of such a system to be the basis of our model. Beyond these key architectural decisions, we allow for a configurable design to enable us to find balanced architectures.

With high latency and a wide connection to the cores, a throughput-oriented approach to computation would benefit more from a ReRAM monolithic memory system. Additionally, the ability to perform many independent, fine-grained fetches means it is well suited to irregular data requests. A GPU could provide this throughput oriented approach, but usually has very little cache; additionally, current GPU programming models focus on coalescing memory requests rather than supporting irregular data fetches. A many-core CPU architecture is throughput oriented and

has a friendlier programming model with the tendency to request data from many disparate memory locations.

Another potential benefit of the monolithically integrated memory is physical proximity to the cores. To take full advantage of this, we co-locate a slice of the main memory with each core. This gives rise to a tiled design, where each tile contains a core and part of the main memory. With the goal of minimizing data movement and creating a very parallel memory system, each tile should also have a memory controller to access the local slice of main memory.

In Chapter 3, we have demonstrated the utility of monolithically integrating the ReRAM over the LLC. This integration means each tile should also have a slice of the LLC, in this case, the L2 cache, co-designed with the main memory. The physical integration of ReRAM and the LLC presents additional opportunities and challenges for their logical design which we explore more in Section 4.2.

For the layout of the tiles, we assume a 2D tiled organization as shown in Figure 4.1. For our initial studies, all tiles will have their own router and are interconnected via a 2D mesh network-on-chip (NoC) with deadlock free e-cube routing. Each tile consists of a core, a private L1 cache (not shown in the figure), a shared L2-main memory slice, a memory controller, and a router.

The cores we focus on are in order and multi-threaded, resembling those found in early chip multi-threaded (CMT) architectures [69, 70], with the addition of single instruction, multiple data (SIMD) execution units. The SIMD units not only execute wide memory operations that fetch contiguous data blocks from memory, they also execute scatter-gather operations from recent ISAs, e.g., AVX-512 [71]. Such scatter-

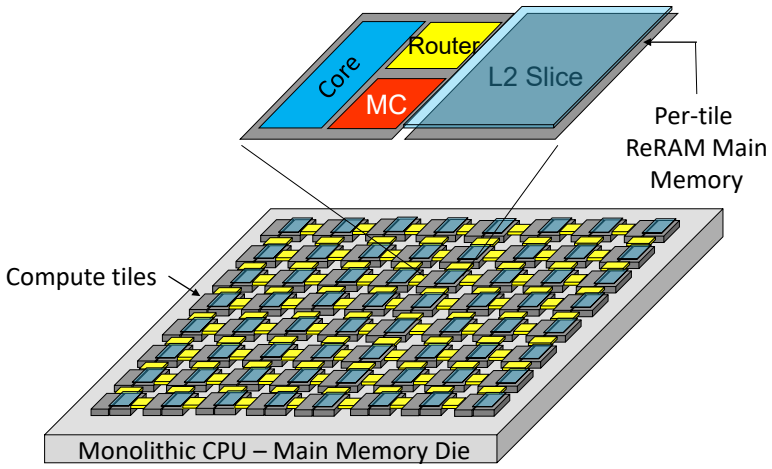


Figure 4.1: Distributing ReRAM main memory across tiles of a many-core CPU.

gather operations are efficiently supported by the fine access granularity of ReRAM. To further increase memory-level parallelism, we also support stride prefetching.

Our many-core CPU employs a 2-level cache hierarchy with private L1s and a physically distributed, but logically shared, L2. Most CMT architectures with shared L2s support directory-based coherence across the private L1s, but early architectures that considered a large number of cores proposed non-coherent cache hierarchies [72]. (GPUs also employ non-coherent private caches). In our work, we focus on a non-coherent hierarchy. Future work might look at how cache coherence would impact the design.

One of the proposed benefits of ReRAM is fine granularity, so our model supports the finest access granularity of practical interest, which we assume to be 8 bytes. Supporting 8-byte accesses benefits applications with sparse memory access patterns. That said, it is also important to support larger cache block transfers from ReRAM, which we have selected to be 64 bytes to match the vector fetch of AVX-512. Assuming 64-byte cache blocks, a cache-line fill would require accessing

8 ReRAM banks together. We would like to model systems with 8-byte fetches, 64-byte fetches, and a multi-grain fetch scheme. Heuristics for multi-grain can include fetching cache blocks for only vector loads or fetching 8 bytes for only gather loads. To handle the multiple request sizes from the memory system, the caches need to be able to store both coarse-grain and fine-grain data blocks. Our caches are sectored and divide their 64-byte cache blocks (i.e., sectors) into 8 subblocks, each holding 8 bytes.

4.2 Cache Organization

In addition to co-designing the cache mats and ReRAM subarrays, monolithic CPU–main memory integration also provides an opportunity to streamline the interface between the LLC and main memory. While 3D integration of ReRAM over cache saves significant area, streamlining the LLC/main memory interface can provide further area savings as well as energy benefits. When co-designing the cache, we output matched the ReRAM banks and cache mats to make some of this streamlining more efficient. This however relies on assumptions about the layout of memory and the addressing scheme. Additionally, the highly parallel monolithic main memory requires a highly parallel cache system to sustain it. We look at the methods to increase cache parallelism and how tightly integrated monolithic main memory may impact them.

4.2.1 Streamlined Interface

Since the main memory and LLC are co-located, they can share components they previously would be unable to share, possibly resulting in further area, delay and energy savings. There are two major system components we consider reducing: internal interconnects and controllers.

Since the LLC and main memory are normally discrete components, there are usually distinct internal interconnects for each of them to facilitate communication from controllers to the memory arrays. But most of the time, the main memory's internal interconnect (for the ReRAM banks) is under-utilized. In the 2 MB cache slice design we have laid out, there are 128 ReRAM banks and currently the best case read latencies are 200 ns, meaning that even when fully accessing all the ReRAM banks, only 128 requests would be sent in 200 ns, a time frame which could service 200 requests.

Given the integrated co-design of ReRAM and cache, the cache access traffic and ReRAM access traffic go to the same place. So, the ReRAM memory controller and cache controller can share a single internal interconnect. This sharing results in modest savings in access time and dynamic energy for the cache. When we looked at the integration impact of ReRAM on the cache such as in Figure 3.27, the "ReRAM" bars represent the single interconnect design while the "Separate Interconnects" bars represent the overhead of requiring two interconnects, one for the cache and one for the main memory. Depending on the configuration, the main benefit of doing so may be to reduce the overall area of the co-design. For 2 MB cache slice, a shared

interconnect reduced the average overall area by an additional 12% from 90% to 78%. A shared interconnect can hurt performance by introducing contention, which is something we explore in the performance simulations in Chapter 5.3.

Another streamlining opportunity is merging the cache controller and memory controller. The cache controller schedules accesses to the cache arrays and keeps track of outstanding misses, while the memory controller schedules the accesses to the memory arrays for those cache misses. The two can be combined by making the cache's miss status holding registers (MSHRs) and the memory controller's scheduling queue (which provides the controller's functionalities) the same structure. This also eliminates communication between the cache and memory controllers, making sharing an interconnect more straightforward, and reducing some latency within the memory system.

If the controllers are merged and the data in the cache mat is in the ReRAM bank above it, we can use a scheme that reduces data movement further. In the case of a cache miss, the request is passed from the cache array to the memory array as soon as the tag check fails without going across the internal interconnect when the ReRAM bank is unoccupied. Similarly, when the data is available from the main memory after the miss, it can immediately be written into the cache array. This saves a traversal of the internal interconnect for every cache miss, and two traversals in the case where it can be serviced immediately by the memory. This also would reduce potential contention introduced by combining the two interconnects.

4.2.2 Addressing

Transferring requests and data immediately from the cache to the main memory or the main memory to the cache assumes that the cache mat caches the data in the ReRAM bank above it. This is a natural assumption that is easily implemented when no virtual memory is present. There is a single address space, the physical one, and the address mapping is 1-to-1 from the cache to the main memory.

When introducing virtual memory and separate address spaces, possibly for protection and multi-programming, the considerations for the addressing scheme can be more complex. This isn't necessarily the case though. The last-level cache design could continue operating purely on physical addresses, translating the virtual address before using it in any caching operations. This is generally how last level caches operate as the L1 cache will often use a physical address to tag data in order to avoid aliasing issues that virtual addressing creates (a physical address being referred to by multiple virtual addresses or a virtual address referring to multiple physical addresses) [73]. Since the translation has already been completed, the physical address will then be used for higher-level caches.

A possible benefit of the virtual memory is the ability to place data local to the tile. If we have no virtual addressing, the data any core may need will be distributed throughout the chip unless the programmer takes particular care to map it to the local tile. With virtual addressing, pages can be placed transparently on the tile that accessed them first, or with a page migration scheme, the most often. Co-locating the memory with the core that uses it reduces communication energy

and latency.

Finally, while the 1-to-1 relationship between the cache and main memory seems like a natural fit, it is not a requirement of the design. If the 1-to-1 relationship constraint were relaxed, we could add another dimension to the area optimization—number of cache mats per subbank. Instead of output matching the cache mats and ReRAM banks, the two would be independent and a single cache mat would likely cache data from multiple ReRAM banks. This is because it is desirable to be able to fetch a full cache line in parallel from multiple ReRAM banks rather than needing multiple serial accesses to a single ReRAM bank; if the cache mat outputs more than 8 bytes of contiguous data, then it will cache data from multiple ReRAM banks. This precludes the optimization where the data is transferred only vertically between cache mats and ReRAM banks but allows for better cache layout optimizations.

In addition to better layout optimizations, it also opens the door to optimizations when cache pressure is asymmetric. If one region of memory is accessed much more frequently than another, a many-to-many relationship between the cache mats and ReRAM banks may allow a larger portion of the working set to stay in the cache than only allowing all data in a ReRAM bank to be cached by a single cache mat.

4.2.3 Cache Parallelism

Because our architecture supports high memory parallelism and bandwidth to address the high latency of ReRAM, throughput is key for performance. With the

cache designs presented, the latency for an access approaches 4 ns at the 32 nm technology node. While this is generally an acceptable latency for the LLC, if a cache slice can only handle one request in that time, it will not meet the throughput requirements. At minimum, a cache slice must be able to handle 128 requests in 200 ns to keep the ReRAM banks occupied, or one request approximately every 1.5 ns. This means that the LLC must be capable of handling requests in parallel.

Cache parallelism can be achieved either via banking or pipelining. Banking allows multiple accesses to occur in parallel in disparate parts of the cache while pipelining allows this parallelism to the same portion of cache. A unified main memory-cache controller and interconnect means considerations must be made for modifying the pipeline to incorporate memory traffic.

A typical cache pipeline has four main stages and sets of resources: decode (d), tag lookup (t), bitline access for writing and reading (w and r), and the MUXing (m) and output (o) stage [74]. There is an additional stage and set of resources when handling a cache miss for the MSHR operations and sending the request to main memory. Many caches will combine the decode and tag lookup stages for faster performance. A read will need to perform the decode and tag operations, then read the data, passing it through the bitline MUXes to the output drivers. A write will perform a read to capture the evicted line and then it will update the tag and pass the data through the MUXes and write it to the cache data array. The miss handling stage is composed of MSHR update (mshr_u), MSHR request (mshr_r), and MSHR output (mshr_o). MSHR update allocates MSHRs and updates them when the cache receives the data from memory. MSHR request sends requests to main

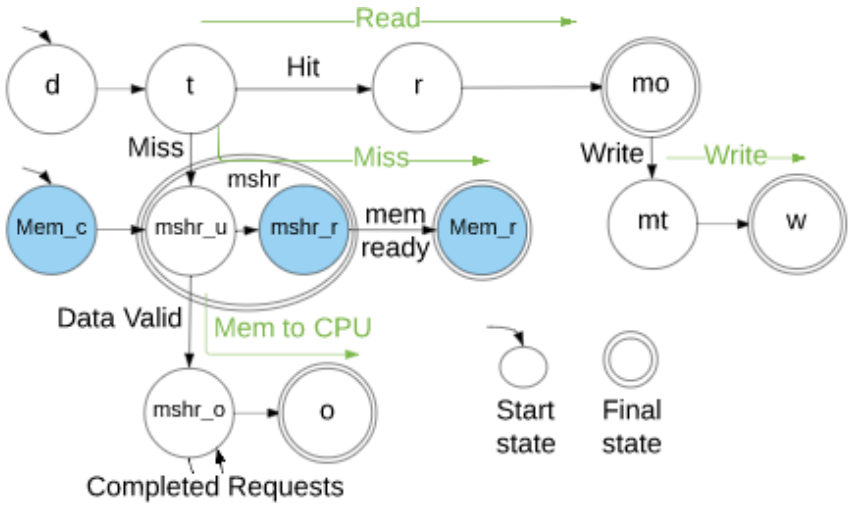


Figure 4.2: State diagram for the cache pipeline. The additional states due to main memory-cache integration are highlighted.

memory. MSHR output sends replies to the CPU. This may take multiple cycles if multiple cores requested the same data as each request will require a separate reply.

The addition of main-memory traffic means that certain resources are now shared in other stages. In Figure 4.2, the stages required to control main memory are highlighted in blue. The main memory request stage (Mem_r) uses the address and data input interconnect required by the decode stage. When the main memory completes the request (Mem_c) and is ready to send the data back, it uses the data output interconnect. The logic involved in the miss handling stage has changed to schedule requests rather than sending them immediately to a memory controller.

Figure 4.3 shows two scenarios one might typically see in the cache pipeline. Figure 4.3(a) shows a write surrounded by read hits; the first two reads after a write require a stall, the first to avoid conflict using the MUXes and the second to avoid conflict in the tag array. Figure 4.3(b) shows what happens when a read miss occurs; if the bank is available (6) after allocating an MSHR, the request will be sent to main

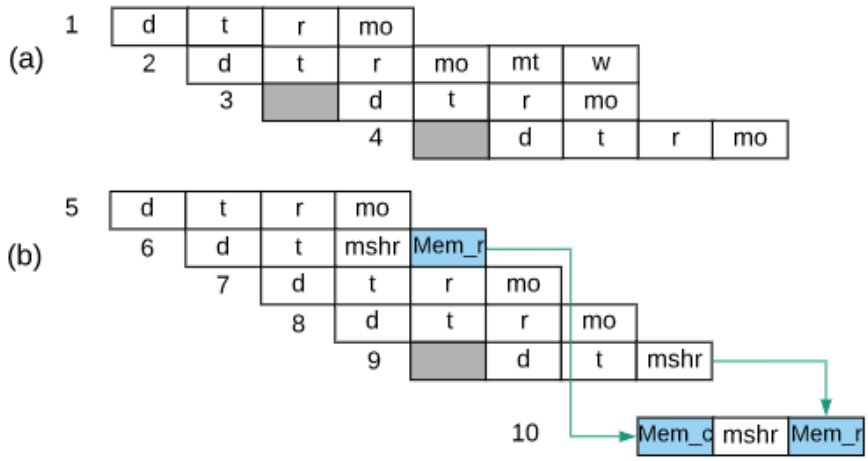


Figure 4.3: Pipeline scenarios. (a) Read hit (1), followed by a write (2) and two read hits (3&4). (b) Read miss to an unoccupied bank (6) and a second read miss to that now occupied bank (9). (10) shows the completion of (6) and sending the request for (9) now that the bank is unoccupied.

memory which will cause a stall to prevent conflict with the address interconnect; if the bank is unavailable (9), the pipeline will only create the MSHR and schedule the request. When a request to main memory completes (10), it will occupy the output interconnect, update the MSHR, and send any pending requests to the bank which is now free. Many more combinations are possible and are modeled in our simulator.

Another bottleneck within the cache may be the MSHR file. Often caches, especially last-level caches, have fairly high hit rates and make little use of the miss path. This also means the MSHR file is usually small and not created for parallel access. The first requirement is to increase the number MSHRS and thus the number of misses the cache can sustain. In simulations, we've found that fewer than one MSHR per ReRAM bank is required. This is because a vector request will occupy a single MSHR but 8 ReRAM banks and writes do not occupy MSHRs. However, bank conflicts and fine-grained accesses mean that often the number of

MSHRs required to not be the limiting factor is not the number of ReRAM banks divided by 8. In simulations, we find that double this value is sufficient.

If using banking to create parallelism in the cache, the MSHR file must also be made parallel. To increase the parallelism of the MSHR file, we can create smaller MSHR files per cache bank or multi-port the MSHR file. Creating smaller MSHR files per cache bank can cause under-utilization of the MSHRs. If one bank has more traffic than others, it will stall when it exhausts its MSHRs, with MSHRs still available in other banks. A single multi-ported MSHR file does not cause this issue but can be quite expensive. This is especially true in our monolithic system as the MSHR file will likely need to be large to accommodate the high number of outstanding requests the memory system needs to maintain a high throughput. Tuck et al. [75] propose a hierarchical design, where a small number of MSHRs are assigned to each bank, and a larger number to the cache slice as a whole. This often allows the parallelism of the MSHR file per bank scheme with the shared capacity of the single file scheme.

4.3 Area and Power Models

We have developed area, dynamic power, and leakage models for all the CPU components in our target architecture. These models will aid us in determining reasonable simulation parameters, such as how many tiles are feasible, and the total power used in the system. In Chapter 3, we use our modified Cacti tool to develop these models for our co-designed last-level cache and ReRAM module. For

the other components of our tile (core, router and memory controller), we use the McPAT tool [76].

McPAT has models for several different types of cores, each including the L1 I- and D-cache. In particular, we use McPAT's Niagra II core because it employs an in-order multithreaded pipeline which is very close to our target architecture. The main modification made was to increase the size of the registers to 512 bits in order to support wide SIMD.

McPAT also has models for a 2-dimensional mesh network router and a memory controller, which we use for the corresponding components in each compute tile of our CPU. McPAT's memory controller model is designed for a conventional discrete memory system in which the memory controller must drive off-chip address and data buses. For our integrated on-die memory system, we exclude the PHY (physical) module from McPAT's memory controller model which contains these large off-chip drivers. We then use this modified version of the memory controller in our simulations.

Our evaluation of monolithic computers considers technology nodes as advanced as the 7nm technology node. Unfortunately, our tools (McPAT and CACTI) only provide area and power parameters down to 22 nm. So, we scaled the results from the tools to acquire the 7nm parameters that we seek. Both McPAT and CACTI provide validated parameters at four technology nodes. McPAT supports the 90 nm, 65 nm, 45 nm, and 22 nm technology nodes; and CACTI supports the 90 nm, 40 nm, 32 nm, and 22 nm technology nodes. We extracted parameters for all the compute components in our CPU–namely, the core, NOC router, memory con-

Table 4.1: Projected Area (mm^2)

Component	Regression Equation	14 nm	7 nm
core	$y = 0.009x^{1.6048}$	0.62	0.20
router	$y = 0.0008x^{1.9325}$	0.13	0.03
memory controller	$y = 0.0721x^{1.0399}$	1.12	0.55
memory controller (Non-PHY)	$0.0219x^{1.1032}$	0.40	0.19
cache	$0.0043x^{1.955}$	0.75	0.19
ReRAM	$0.007x^{1.9986}$	1.25	0.32
cache/ReRAM	$0.0073x^{1.9986}$	1.43	0.36
cache/ReRAM 2 HTrees	$0.008x^{1.9971}$	1.56	0.39

troller, and integrated ReRAM-LLC slice–at each technology node supported by the tool used. Then, we performed regressions on the extracted values and extrapolated the results to 14 nm and the more aggressive 7 nm.

4.3.1 Area

To determine how many big of a CPU and main memory could realistically be placed on a die, we first compute the size of each tile at our chosen technology nodes. Table 4.1 reports the extrapolated parameters for area and Figures 4.4–4.11 show the values per technology node and the final regressions.

Most of our evaluation assume the 14 nm technology node and the co-designed cache/ReRAM with an integrated interconnect. Each of the tiles in this system is composed of the core (0.62 mm^2), the router (0.13 mm^2), the memory controller without the phsyical I/O components (0.40 mm^2) and our co-designed and streamlined main memory-cache slice (1.43 mm^2). This means these tiles are 2.58 mm^2, with 256 tiles fitting on a 660.5 mm^2 die. This is a large die, but similar to Intel's

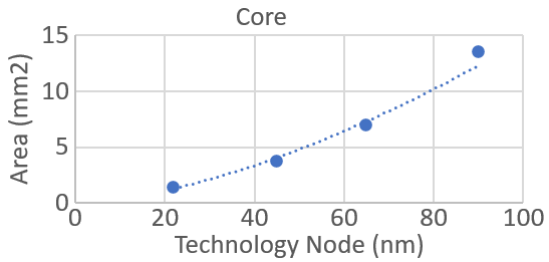


Figure 4.4: Area regression for the core from McPAT.

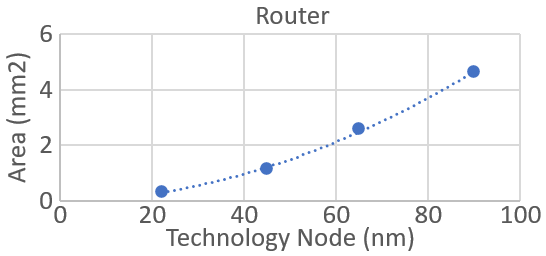


Figure 4.5: Area regression for the router from McPAT.

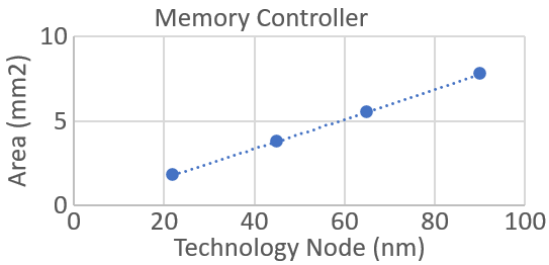


Figure 4.6: Area regression for the memory controller from McPAT.

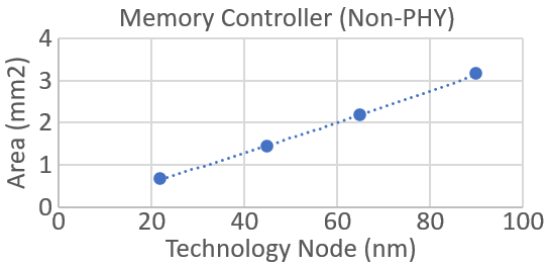


Figure 4.7: Area regression for the memory controller without the PHY component from McPAT.

Knights Landing's 683 mm^2 die [77]. This would provide a 256 GB main memory system.

If we instead wanted to model the cache and ReRAM without any integration, we would replace the combined cache/ReRAM unit with the separate cache (0.75 mm^2) and ReRAM (1.25 mm^2). This would result in a tile that is 3.15 mm^2, 22.1% larger than our co-designed cache, and only 210 tiles could be integrated in the same area. Chapter 5.3.8 attempts to model the performance impact of this.

4.3.2 Dynamic Power

For the core, NOC router, and memory controller components, McPAT provides peak power results assuming an average amount of activity in these compo-

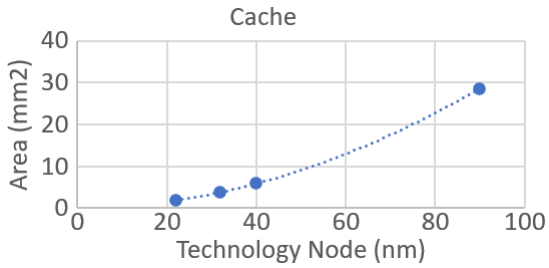


Figure 4.8: Area regression for the cache from CACTI.

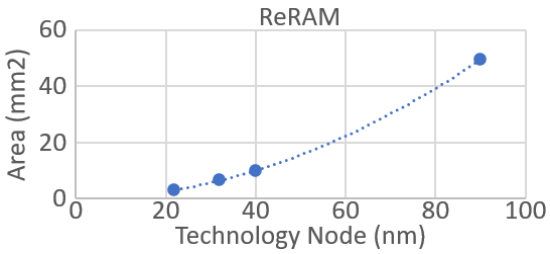


Figure 4.9: Area regression for the ReRAM slice.

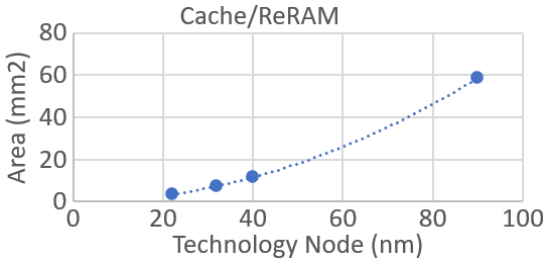


Figure 4.10: Area regression for the co-designed ReRAM/Cache slice from CACTI.

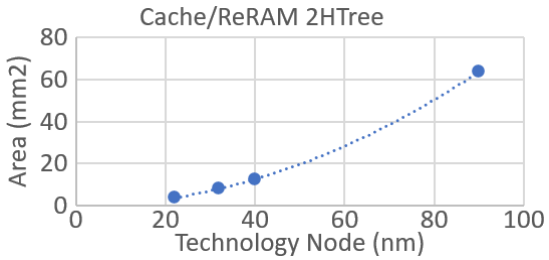


Figure 4.11: Area regression for the co-designed ReRAM/Cache slice with separate interconnects from CACTI.

nents. In contrast, our modified CACTI tool provides the energy per access for the cache. Energy per memory access rather than peak power is more useful for our architecture-level simulator since it simulates the activity on a per-cycle basis, and then sums the resulting energies to derive the average dynamic power. Whereas, we have much simpler models for our cores, routers, and controllers, so we make use of McPAT's estimations and execution time to calculate dynamic energy in the non-memory components of the system.

Table 4.2 reports the extrapolated parameters for dynamic power/energy and Figures 4.12–4.18 show the values per technology node and the final regressions. The regression graphs for the McPAT tool, Figures 4.12–4.13, show the results for average dynamic power, and Table 4.2 reports the final average power numbers extrapolated to 14nm and 7nm. Figures 4.16–4.18, which are generated by the modified Cacti

Table 4.2: Projected Dynamic Power

Component	Regression Equation	14 nm	7 nm
McPAT: Peak Dynamic Power (W)			
core	$y = 0.0009x^{1.811}$	0.11	0.03
router	$y = 0.0009x^{1.6807}$	0.08	0.02
memory controller	$y = 0.003x^{1.336}$	0.10	0.04
memory controller (Non-PHY)	$y = .0005x^{1.6159}$	0.04	0.01
Cacti: Dynamic Read Power (nJ/access)			
cache	$y = 0.0001x^{1.7924}$	0.011	0.003
cache/ReRAM	$y = 0.0002x^{1.7443}$	0.02	0.01
cache/ReRAM 2 HTrees	$y = 0.0002x^{1.7393}$	0.02	0.01

Tool, show energy per access; Table 4.2 reports the final 14nm and 7nm values for energy per access to the last-level cache.

Finally, we do not currently have a typical access energy for ReRAM. It is a newer technology, with many variations in composition, and currently is often targeted as a storage class memory (SCM) rather than as main memory. The write energy reported in the literature ranges from 1.3 pJ/bit [26] and 1.6 pJ/bit [28] to 53.2 pJ/bit [78] and 65 pJ/bit [79]. Researchers are currently looking to re-target the ReRAM as a main memory which will affect its latency, access energy and wear-out characteristics. Due to this uncertainty, when calculating energy, we will use a range for the possible ReRAM values.

4.3.3 Leakage

For the area and dynamic power / energy regressions (Figures 4.4–4.18), all four technology nodes available in McPAT and CACTI are used in the regression.

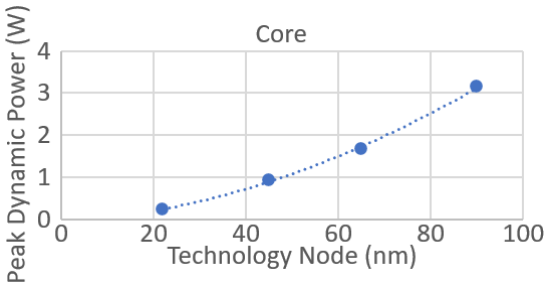


Figure 4.12: Peak dynamic power regression for the core from McPAT.

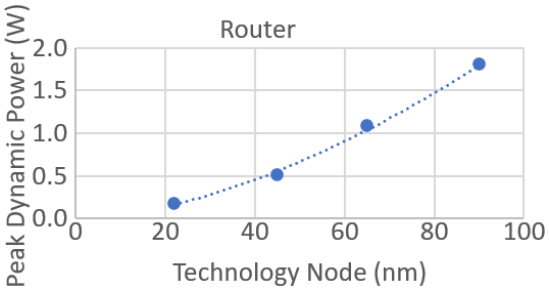


Figure 4.13: Peak dynamic power regression for the router from McPAT.

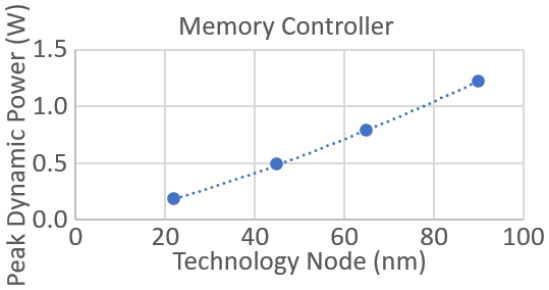


Figure 4.14: Peak dynamic power regression for the memory controller from McPAT.

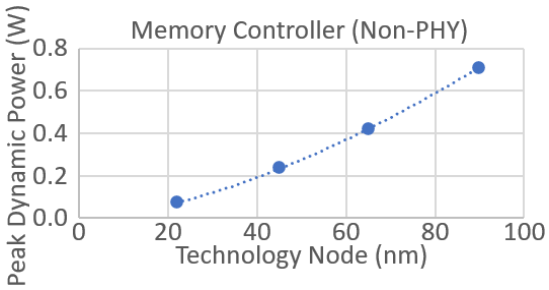


Figure 4.15: Peak dynamic power regression for the memory controller without the PHY component from McPAT.

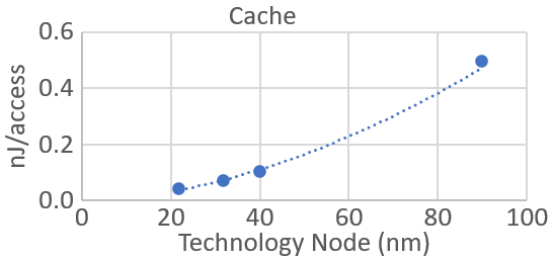


Figure 4.16: Dynamic access energy regression for the cache from CACTI.

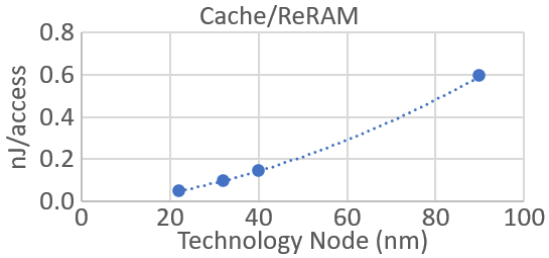


Figure 4.17: Dynamic access energy regression for the co-designed ReRAM/-Cache slice from CACTI.

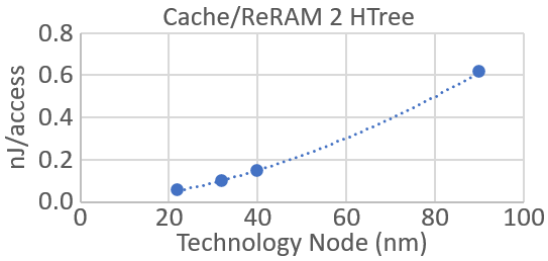


Figure 4.18: Dynamic access energy regression for the co-designed ReRAM/-Cache slice with separate interconnects from CACTI.

Table 4.3: Projected Leakage Power (W)

Component	Regression Equation	14 nm	7 nm
core	$y = 0.408x^{0.7482}$	0.29	0.17
router	$y = 0.0916x^{0.3461}$	0.23	0.18
memory controller	$y = 0.0059x^{0.7244}$	0.04	0.02
memory controller (Non-PHY)	$y = 0.0026x^{0.6998}$	0.02	0.01
cache	$y = 0.0018x^{1.18751}$	0.25	0.07
cache/ReRAM	$y = 0.0015x^{1.9748}$	0.28	0.07
cache/ReRAM 2 HTrees	$y = 0.0015x^{1.9714}$	0.27	0.07

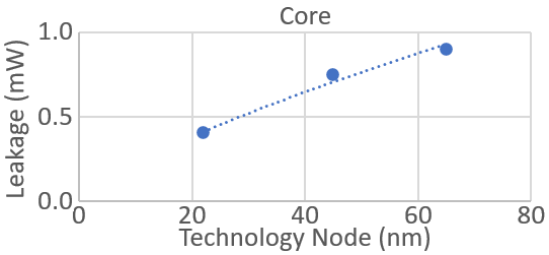


Figure 4.19: Leakage power regression for the core from McPAT.

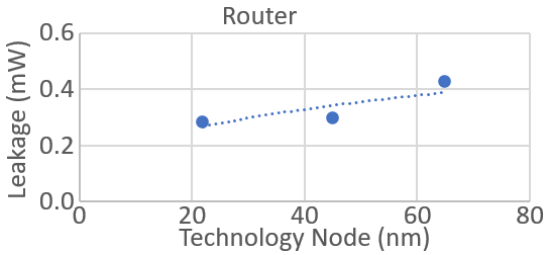


Figure 4.20: Leakage power regression for the router from McPAT.

However, for the leakage regression (Figures 4.19–4.25), we omit the 90nm technology node. The leakage increases from 90nm to the next technology node (65nm for McPAT and 40nm for CACTI) opposing the observed trend in the other technology nodes which results in a poorly fit regression that is not useful for extrapolation to 14nm and 7nm. Due to this, we only used the last 3 technology nodes for the regression and extrapolation. It is possible static power will have an additional change like this as we go below 22nm. Table 4.3 reports the extrapolated parameters for leakage power.

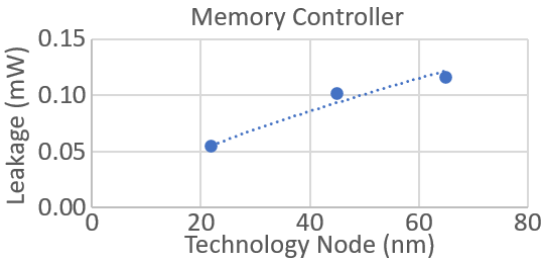


Figure 4.21: Leakage power regression for the memory controller from McPAT.

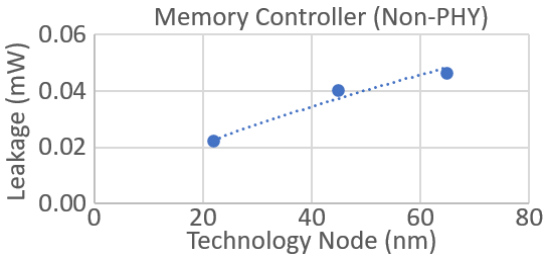


Figure 4.22: Leakage power regression for the memory controller without the PHY component from McPAT.

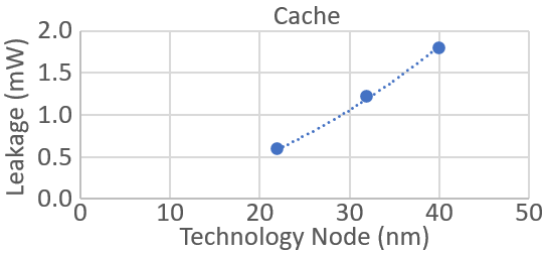


Figure 4.23: Leakage power regression for the cache from CACTI.

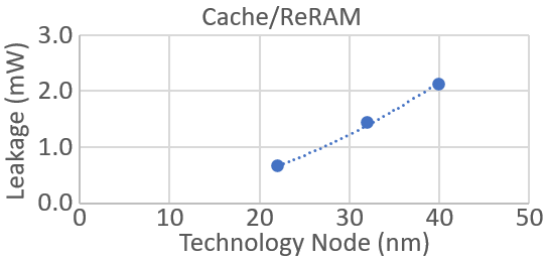


Figure 4.24: Leakage power regression for the co-designed ReRAM/Cache slice from CACTI.

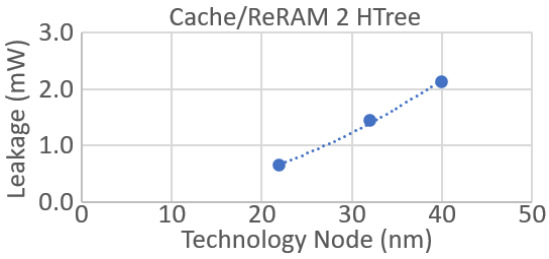


Figure 4.25: Leakage power regression for the co-designed ReRAM/Cache slice with separate interconnects from CACTI.

4.4 Conclusion

In this chapter, we have developed a many-core architecture that could support a monolithic main memory system. The architecture focuses on throughput-oriented computation and takes advantage of ReRAM's parallelism and fine granularity. The architecture is a tiled many-core design with each tile containing a core, L1 cache, router, L2 slice monolithically integrated with ReRAM main memory and a memory controller.

The physical integration of the cache and main memory creates opportunities for streamlining the interface between the two. Instead of two separate interconnects, a single one can be used for both memory and cache traffic. A single controller and structure can be used to track outstanding cache misses/requests to main memory. The data could also be transferred vertically directly between the cache and main memory as they are now co-located and a single controller coordinates both.

The monolithic main memory system needs high throughput to have good performance, meaning the cache must be parallel to keep up. This can be accomplished with banking or pipelining. With the pipeline scheme, the shared interconnect means that new states to communicate with the main memory need to be incorporated. With the banking scheme, the MSHR file needs to be designed to maintain the parallelism when a high number of misses occur that may not be balanced between cache banks. A hierarchical MSHR file is a good option for heavily banked caches.

Using McPAT and our modified version of Cacti, we show that 256 tiles with in-order cores, 4-way hardware multi-threading and wide SIMD, and a last-level

cache/main memory slice with 2 MB SRAM and 1 GB ReRAM could feasibly fit on a KNL-size die. We will use these calculations to inform our simulation parameters in the next chapter.

Chapter 5: Simulation

5.1 Simulator Architecture

We created a simulator to model our many-core CPU with an on-die main memory system. A high-level block diagram is shown in Figure 5.1. Similar to recent simulators of many-core CPUs with a very large number of threads [80, 81, 82, 83], we use an Intel Pin-based front-end [84] to feed a parallel instruction trace to the cycle accurate back-end. The main difference is that for the front-end, we employ Intel's Software Development Emulator (SDE) [9]. SDE can execute x86 binaries with 1000s of threads. It can also emulate AVX-512 instructions that support scatter-gather operations on processors that lack that ISA extension.

Unlike previous multi-core simulators, ours sacrifices simulation speed in favor of cycle-accuracy with respect to the memory system. Since our research is primarily interested in the memory system, the typical memory latency cannot be assumed and then attempted to be corrected at a later synchronization point, the approach taken by other many-core simulators. This is the impetus for developing our own manycore simulator.

Cores. The front-end executes all threads in SDE and maps them to each of the back-end cores. Cores are single issue, in order and multithreaded with a

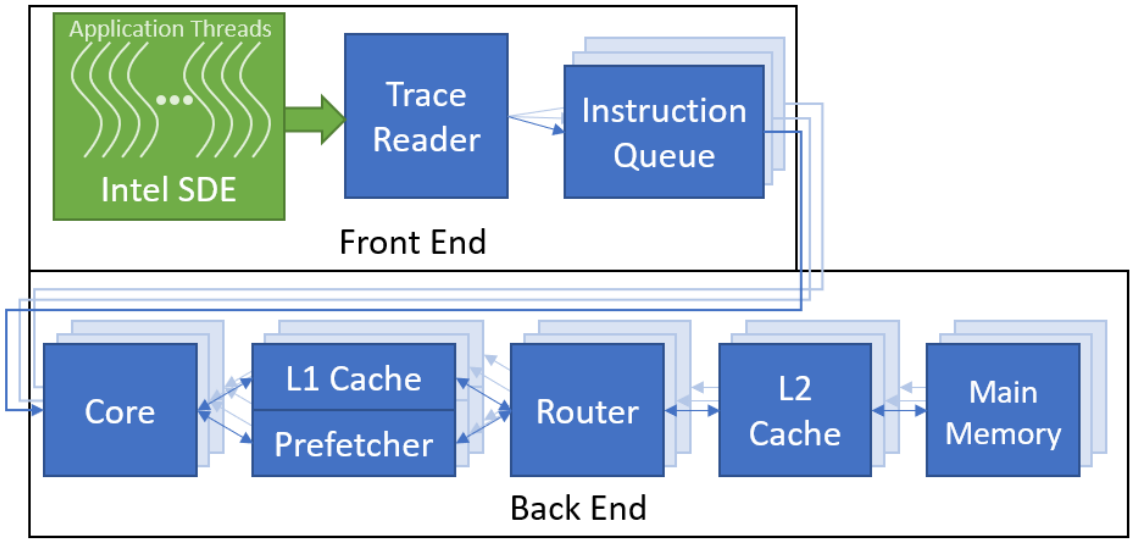


Figure 5.1: Overview of simulator architecture.

configurable context switching overhead. The cores employ a simple one cycle per instruction (one-IPC) model and stall threads whenever they encounter a cache read miss. When encountering a scatter-gather instruction, the core will perform each access on consecutive cycles and the thread will only stall once all read requests have been sent.

For the benchmarks we have selected, the cores are frequently idle, so overheads from branch mispredicts or extra computation time required for floating point computations are not included. Additionally, it's been shown that one-IPC simulation models work well for relative performance when focused on homogeneous cores working on homogeneous workloads [80]. All our simulations meet these requirements. If more computationally intense benchmarks, heterogeneous workloads or cores were included, a more complex core model would be required.

L1 Cache. The size and set associativity of the L1 cache is configurable. Hardware prefetching can be enabled and is performed at the L1 cache level via a

2-delta stride table with a configurable number of entries. After stride detection, data prefetching occurs into configurable depth prefetch buffers. The L1 cache is flushed when the core reaches a barrier to support the non-coherent cache hierarchy. When a cache miss occurs, the L1 cache uses the address to calculate the coordinates of the L2 slice where the data is stored and sends a request. If the network buffer is full, it will stall the core and attempt to send the request again during the next cycle.

Network on Chip. The tiles are interconnected via a two-dimensional mesh network. The number of nodes in either direction is configurable but must be a power of 2. The NoC carries the traffic between the private L1s and the L2 slices on L1 cache misses. It employs dimension-ordered routing and supports either 2 physical channels or 2 virtual channels (one for requests and one for replies) to ensure it is deadlock free. The routing and request information is contained in an 8-byte header sent with every packet. The width of the NoC is configurable in 8-byte increments.

L2 Cache. The L2 cache's size, associativity, and number of independent banks is configurable. Pipelining can be enabled or not. The main memory can be integrated, modeling our co-designed LLC/ReRAM main memory module and streamlined LLC/main memory interface, or it can have a separate interface and interconnects from the cache. The L2 cache is sectored to allow cache-miss fills at either 8-byte or 64-byte granularity, as determined by a bit in the request header. Each cache slice has a configurable number of MSHRs. If pipelined, the cache follows the pipeline presented in Section 4.2.3, with each stage handling a single request and

requiring a cycle to execute. Scheduling ReRAM requests takes place during the MSHR stage.

Main Memory. If the LLC/main memory interface is not streamlined, the L2 cache is connected directly to a ReRAM controller. The ReRAM controller has a queue with a configurable number of entries for each bank but does not otherwise re-order requests. The number of ReRAM banks, their read latency, and their write latency are all configurable. We faithfully model all bank conflicts and queuing at the controllers.

If a DRAM system is being modeled, an additional two network channels are added from the L2 cache to the memory controllers. The number of DRAM controllers and their placement on the network is configurable. The main memory portion of the simulator is replaced with DRAMSim3 [85] which accurately models all DRAM components.

5.2 Benchmarks

The benchmarks used in our performance evaluation are listed in Table 5.1. The benchmarks selected are memory intensive, highly parallel with thread-level parallelism and SIMD parallelism. Half of the benchmarks are irregular graph kernels and half are streaming computations. The graph kernels are from CRONO [86] and the streaming computations, other than DAXPY, are from Rodinia [87]. We implemented DAXPY which computes $y[i] = a * x[i] + y[i]$. We selected benchmarks which execute fully parallel loops, with dependences occurring only across barriers

Graph Kernels		
All Pairs Shortest Path (APSP)	scale = 22	1.4
Betweenness Centrality (BC)	scale = 22	1.5
Page Rank (PR)	scale = 22	2.1
Single Source Shortest Path (SSSP)	scale = 22	1.8
Streaming Computations		
daxpy	1.47B elements	1.5
K-means (KM)	1M points, 32 features	1.0
Nearest Neighbor (NN)	204.8M hurricanes	1.3
Pathfinder (PF)	15M cols, 100 rows	2.4

Table 5.1: Benchmark names, input sizes, and number of instructions simulated (in billions).

to allow us to manually manage coherence.

The second column of Table 5.1 specifies the inputs for each benchmark. For the graph kernels, we created an input graph using SSCA2 [88] which is based on the Recursive MATrix (R-MAT) scale-free graph generator [89]. The number of vertices in the generated graph is 2^{22} (i.e., scale=22). For the Rodinia benchmarks, the input sizes are scaled-up versions of the inputs provided with the benchmarks [87] to match the high degrees of parallelism in our experiments.

The final column is the number of instructions, in billions, we simulated for each benchmark. For the streaming benchmarks, these represent the entire parallel region. For the graph kernels, the instruction counts represent a portion of the parallel region. All benchmarks exhibit the same behavior throughout the parallel region, so we expect the 4 partially simulated benchmarks to still yield representative results.

All the kernels were explicitly parallelized to create threaded code. The threaded code was then vectorized by hand using AVX-512 intrinsics to generate

SIMD instructions. Unit-stride array traversals occur in all of the benchmarks and were converted to packed vector load/store instructions. In the graph kernels, memory indirection through edge lists is ubiquitous, and were converted to scatter-gather memory instructions. There is also opportunity for scatter-gather in one of the streaming benchmarks, NN. NN accesses 16 bytes every 64 bytes; rather than fetching the full 64-byte cache block, we use gather operations to fetch only the required 16 bytes.

Figures 5.2 and 5.3 show the vectorization process for a streaming computation, using DAXPY as an example. While modern compilers can often manage this optimization for simple loops like the one in DAXPY, we used AVX-512 intrinsics to explicitly insert the AVX-512 instructions. In the intrinsics, the prefix `mm512` indicates its operating on 512-bit operands; the suffix `epi64` indicates the vector is composed of packed 64-bit signed integers, while `pd` indicates it's composed of packed doubles. The central part of each intrinsic is often recognizable as a typical assembly instruction.

The first 6 lines are identical between the two implementations and are just setting up the problem. Lines 8 and 9, in the vectorized version, create local copies of the points to the global x and y arrays. Without these lines, the compiler must request them from memory every time as another thread could alter the pointer values; this optimization would also benefit the threaded version. Line 11 declares the variables used in AVX-512 operations; each variable is 512 bits, allowing 8 doubles to be operated on with a single instruction. Line 12 broadcasts the double *alpha* to all elements of $zmm2$ so it can be used for 8-wide operations. Line 14

```
1  void *Pth_daxpy(void* rank) {
2    long my_rank = (long) rank;
3    int local_n = n/thread_count;
4    int my_first_i = my_rank*local_n;
5    int my_last_i = my_first_i + local_n;
6    int i;
7
8    for (i = my_first_i; i < my_last_i; i ++) {
9      y[i] += alpha*x[i];
10   }
11 }
```

Figure 5.2: Parallelized DAXPY implementation.

```
1  void *Pth_daxpy(void* rank) {
2    long my_rank = (long) rank;
3    int local_n = n/thread_count;
4    int my_first_i = my_rank*local_n;
5    int my_last_i = my_first_i + local_n;
6    int i;
7
8    double* x_l = x;  //create local address pointer to x
9    double* y_l = y;  //create local address pointer to y
10
11   __m512d zmm0,zmm1,zmm2;           //declare 512 bit variables
12   zmm2 = _mm512_set1_pd(alpha);     //zmm2 = alpha
13
14   for (i = my_first_i; i < my_last_i; i += 8) {
15     zmm0 = _mm512_loadu_pd(x_l+i);  //zmm0 = x[i]
16     zmm1 = _mm512_loadu_pd(y_l+i);  //zmm1 = y[i]
17     zmm0 = _mm512_mul_pd(zmm0,zmm2);//zmm0 = alpha*x[i]
18     zmm1 = _mm512_add_pd(zmm1,zmm0);//zmm1 = y[i] + alpha*x[i]
19     _mm512_storeu_pd(y_l+i,zmm1);   //y[i] = y[i] + alpha*x[i]
20   }
21 }
```

Figure 5.3: Parallelized and vectorized DAXPY implementation.

```
1  void *RandomAccessUpdate(void * tid) {
2    u64Int i;
3    u64Int ind;
4    ind = starts(loops * (u64Int)tid);
5
6    for (i=0; i<loops; i++) {
7      ind = (ind << 1) ^ ((s64Int) ind < 0 ? POLY : 0);
8      Table[ind & (TableSize-1)] ^= ind;
9    }
10 }
```

Figure 5.4: Parallelized GUPS implementation.

is equivalent to line 8 in Figure 5.2; the only difference is it advances 8 elements per loop rather than 1. The loop body breaks the computation down to individual instructions. Lines 15 and 16 load 8 doubles starting at $x[i]$ and $y[i]$. Line 17 multiplies each of the 8 elements of *alpha* and $x[i]$; line 18 adds the results to each of the 8 elements of $y[i]$. Finally, the result is stored is the 8 consecutive elements pointed to by $y[i]$. While this is more lines of code, the compiler would break down line 9 in Figure 5.2 into similar instructions.

Figures 5.4 and 5.5 show the vectorization process for a kernel with indirect memory references which can take advantage of the gather and scatter operations. We use GUPS as an example to highlight a benchmark which is only reliant on these indirect memory references.

Lines 10 through 16 are setting up the starting indices for all 8 elements, similar to line 4 in the non-vectorized version. Theoretically, this could be done in parallel as well, but it is done serially as it represents such a small part of the processing time. Lines 17 through 23 are setting up the constants (*ts*, *poly*, 0 and 1) in a similar way to line 12 in Figure 5.3. Again, the loop body breaks down

```
1  void *RandomAccessUpdate(void *tid) {
2    u64Int i;
3    u64Int *start;
4    64Int tSize = TableSize - 1;
5    __mmask8 cmp;
6    __m128i one;
7    __m512i rans, poly, ts, ind, tab, zero, ind_ts;
8
9    /* Initialize starting indices */
10   start = (u64Int *)_mm_malloc((sizeof(u64Int)*8), 64);
11   for (i=0; i<8; i++)
12     start[i] = starts(loops * (((u64Int)tid*8) + i));
13
14   //load starting 8 64 bit integers into ind
15   ind = _mm512_load_epi64(start);
16   _mm_free( start );
17   //copy POLY 8 times to fill poly
18   poly = _mm512_set1_epi64(POLY);
19   //copy TableSize - 1 8 times to fill ts
20   ts = _mm512_set1_epi64(tSize);
21   //create zero and one
22   zero = _mm512_xor_epi64(zero, zero);
23   one = _mm_cvtsi64_si128(1);
24
25   for (i = 0; i < loops; i++) {
26     //ind = (ind << 1) ^ ((s64Int) ind < 0 ? POLY : 0);
27     inds = _mm512_sll_epi64(ind, one);          //inds = ind << 1
28     cmp = _mm512_cmplt_epi64_mask(ind, zero);  //cmp = (signed)ind < 0
29     ind = _mm512_maskz_mov_epi64(cmp, poly);   //ind = (cmp) ? POLY : 0
30     ind = _mm512_xor_epi64(ind, inds);          //ind = ind XOR inds
31
32     //Table[ind & (TableSize-1)] ^= ind;
33     ind_ts = _mm512_and_epi64(ind, ts);  //ind_ts = ran & (tablesize-1)
34     //tab = Table[ind_ts]
35     tab = _mm512_i64gather_epi64(ind_ts, Table, 8);
36     tab = _mm512_xor_epi64(ind, tab);    //tab = tab ^ ran
37     //Table[ind_ts] = tab
38     _mm512_i64scatter_epi64(Table, ind_ts, tab, 8);
39   }
40 }
```

Figure 5.5: Parallelized and vectorized GUPS implementation.

the complex lines of code into simpler assembly level instructions. Line 28 creates an 8-bit mask, and in line 29, `_mm512_maskz_mov_epi64` will copy values from *poly* corresponding to elements with a 1 in the mask, and zero out other elements. Line 33 calculates the 8 indices to access this loop. Line 35 uses these indices to gather (load) the eight elements from across memory in a single instruction. Line 38 XORs the elements with *ind*. Finally, line 38 uses the calculated indices to scatter (store) all eight elements across the memory in a single instruction.

In addition to parallelization and vectorization, in the streaming computations, we were able to partition the arrays which exhibited unit-stride traversals such that the array portion operated upon by each core is allocated in the core's local cache slice and ReRAM memory. However, the graph kernels exhibited large amounts of irregular accesses, accessing data across a large portion of the memory space, preventing them from being partitioned this way.

While this was an involved process for our benchmarks, we believe this will be accomplished transparently by many compilers in the near future. Compilers already support automatic vectorization to some degree [90], and researchers continue to develop techniques to improve auto-vectorization [91, 92, 93]. With the continued inclusion of SIMD units and the AVX-512 instruction set becoming standard in Intel's consumer CPUs, it is likely to keep improving.

Cores	256, 4-way multi-threaded
Clock rate	2 GHz
L1 Cache	32 KB, 4-way, 1 cycle
L2 Cache Slice	256 KB, 8-way
Stride Prefetcher	32-entry stride table
Unified Controllers	256 (1 per core), 64 MSHRs
ReRAM banks	32,768 (128 per tile)
ReRAM read/write latency	200/400 ns
On-chip Network	16 x 16, 2D-Mesh
Network Channels	4 x 64 bytes

Table 5.2: Baseline simulation parameters for the experiments.

5.3 Simulation results

We have used the simulator to do many parameter sweeps for our tiled monolithic system. We have also used it to establish the performance of a DRAM High Bandwidth Memory 2 (HBM2) memory system with a similar compute architecture. This section will present many of these results. For all results, the performance metric is instructions per cycle (IPC).

Table 5.2 lists the simulation parameters we use as a baseline for our simulations. We try varying many of these parameters, but unless otherwise mentioned, they will have the value specified in Table 5.2. The simulated L2 slices are smaller than the slices designed in Section 3.2 to account for the much smaller data inputs we are able to simulate. The heuristic selected for fetch granularity is only gather requests will fetch 8 bytes, vector requests and single load/stores will be done at the cache block granularity.

5.3.1 DRAM Baseline

One of the first things to establish is how these benchmarks perform in a more traditional system. We selected the HBM2 memory system with either 4 stacks or an aggressive 8 stacks. This memory is fairly configurable and allows hardware designers to decide how many channels each stack will have, up to 8, and the fetch granularity from 32 bytes to 1024 bytes.

Since we have created a very parallel compute system, we focus on configurations with the most amount of memory parallelism available. This means that we maximize the number of channels for each stack. We look at three fetch widths. The smallest fetch width possible is 32 bytes, which favors the graph benchmarks; 64-byte fetch is a fairly standard fetch width; finally, we use 128-byte fetch width, with half the number of controllers, which favors streaming computations. For the 32-byte and 64-byte fetch widths, each stack has 8 controllers; for 128-byte fetch widths, each stack has 4 controllers.

Figure 5.6 shows the performance of the HBM2 memory system for the selected configurations. The 8-stack configuration has roughly double the performance of the 4-stack configuration. As expected, the smaller fetch width with more controllers benefits the graph kernels whereas the streaming computations benefit from having a larger fetch width, even if it has fewer controllers. These simulations give us a baseline to compare our monolithic system against. We will compare the graph kernels against the 32-byte fetch and the streaming computations against the 128-byte fetch unless otherwise noted.

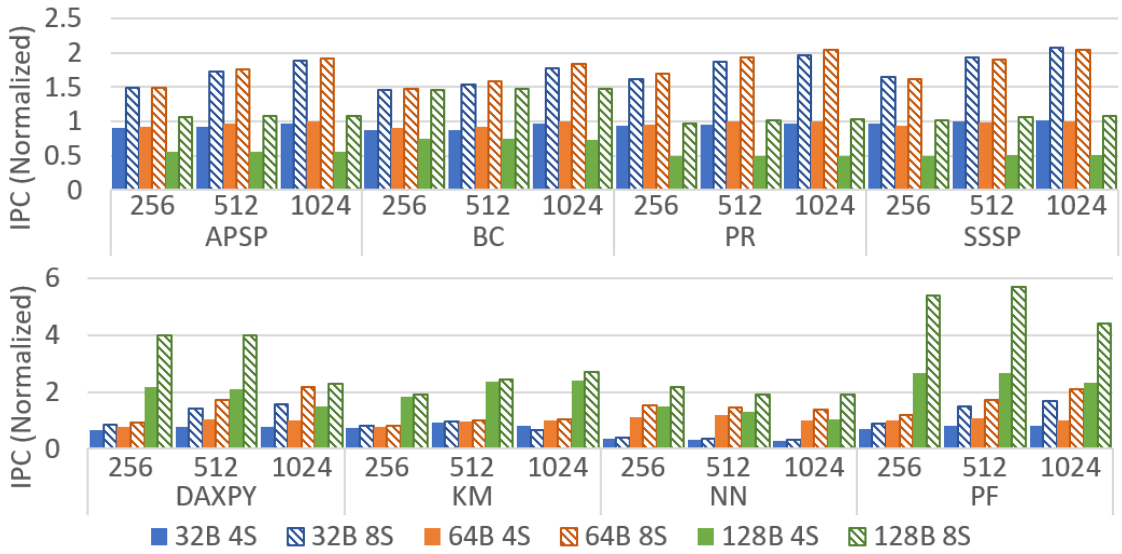


Figure 5.6: IPC of each benchmark as the number of stacks and the fetch width of the HBM2 system are varied, normalized to the 1024 thread, 4 stack, 64-byte fetch case.

5.3.2 Threads

We start from the baseline configuration detailed in Table 5.2 and vary the number of threads in the system to gauge the amount of parallelism required to compete with the DRAM system. We have simulated 1 thread per tile, 2 threads per tile, and 4 threads per tile for each of the benchmarks. The results are presented in Figure 5.7, compared against the DRAM systems.

For the graph benchmarks, the monolithic memory system improves in performance as the number of threads increases. At 256 threads, the ReRAM outperform 4-stacks of HBM2 on the graph benchmarks by 2.4x and 8-stacks of HBM2 by 1.5x. By 1024 threads, this increases to a performance increase of 5.3x and 2.7x, respectively. SSSP has the best relative performance, outperforming 4- and 8-stack HBM2 by 6.9x and 3.2x, respectively.

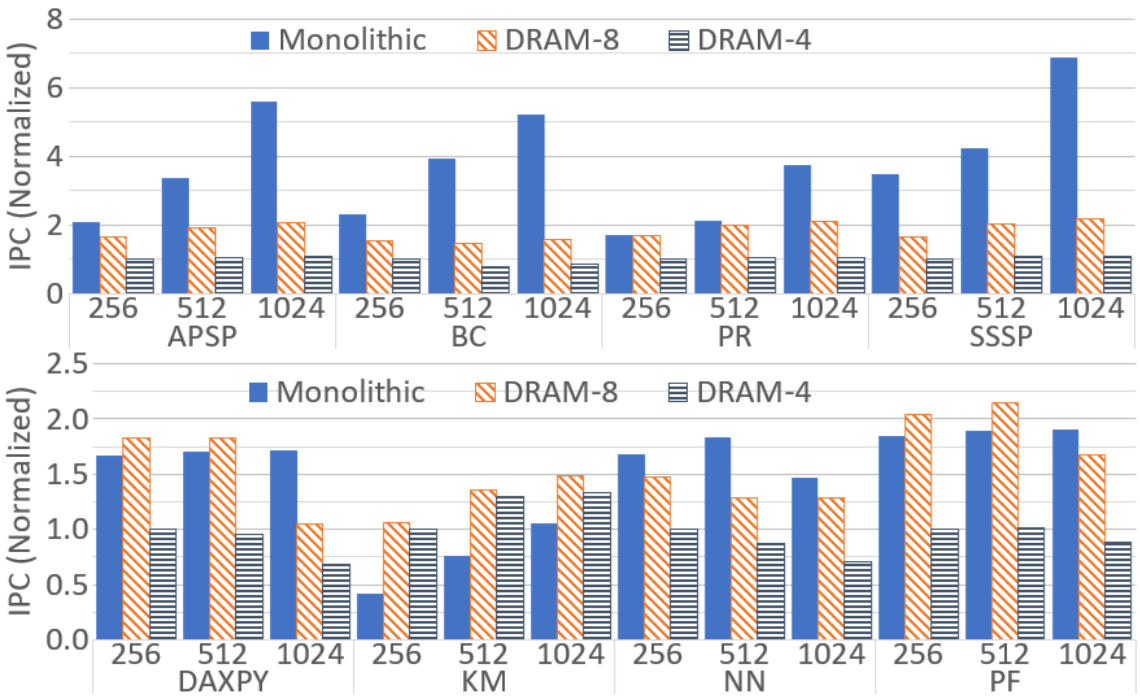


Figure 5.7: IPC of each benchmark as threads are varied for the given memory system, normalized to the 256 threaded "DRAM-4" case.

The higher amount of thread-level parallelism (TLP) increases the memory parallelism and allows some of the latency to be hidden by other threads doing work. The DRAM systems are overwhelmed by the number of requests, even at the smallest thread count, and so their performance does not scale. This is one of the worst scenarios for DRAM as it often relies on row buffer locality to achieve its maximum performance and there is little to no locality in these benchmarks.

The streaming computations are generally less sensitive to the number of threads. The prefetcher works very well for streaming benchmarks and generates enough memory parallelism with less TLP. This is not the case for KM, which is the least memory intensive and tends to need more threads to hide the higher latencies. NN's performance actually suffers as the number of threads increases. This is due to very high contention as only a quarter of the banks are being used due to NN's

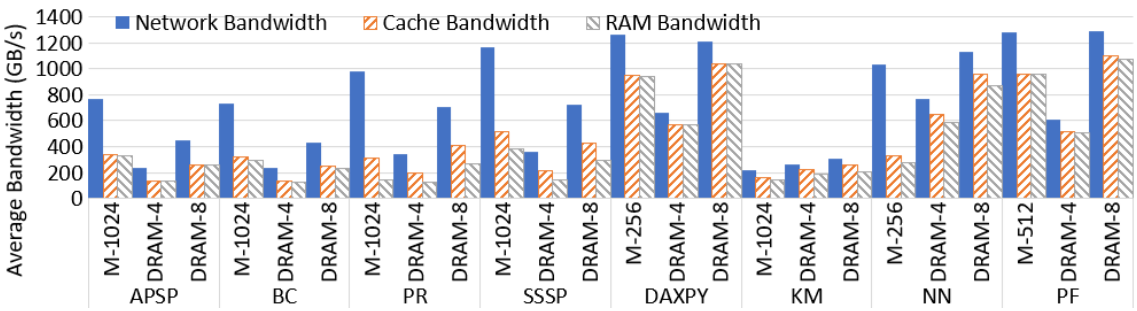


Figure 5.8: Average Bandwidth.

stride. For the best performing thread count, the ReRAM system's performance is 1.4x that of the HBM2 4-stack's performance and 0.9x for 8-stacks of HBM2.

The average bandwidth of the network, cache, and main memory are shown in Figure 5.8. We only show the thread count for which DRAM performs the best per benchmark. The cache and RAM bandwidths are very similar for most benchmarks, as L2 cache hits are rare. The exceptions are PR and SSSP, which exhibit a degree of locality and so benefit from the LLC, resulting in their cache bandwidth being 2.2x and 1.3x larger than their RAM bandwidth, respectively. The issue of interleaving streams is evident for the discrete systems, as the DRAM only reaches 63% of its peak bandwidth in the best case for the streaming benchmarks.

The network bandwidth includes a request header and address, increasing its bandwidth usage over the cache bandwidth. This is most evident in benchmarks that use scatter-gather requests, as each packet will have a larger ratio of header information to data. For gather reads, there will 3x more header information than data; whereas, for full cache block reads, there is 2.7x more data than header information. This results in benchmarks with fine grained requests having network bandwidth that is 2.6x larger than the cache bandwidth, compared to 1.3x for the

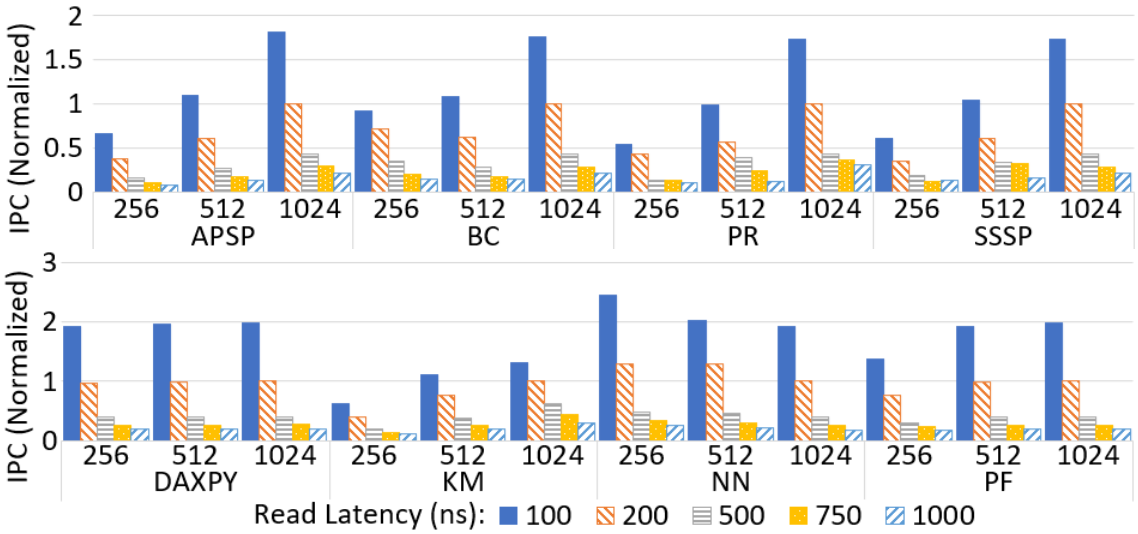


Figure 5.9: IPC of each benchmark as the read and write latency is varied for the monolithic ReRAM memory system, normalized to the 1024 threaded, 200 ns case. The write latency is always 2x the read latency.

benchmarks which perform no fine-grained accesses. This is similarly true for the DRAM. Since we use 32-byte fetches for graph benchmarks, the network bandwidth is 1.7x larger than the cache bandwidth; for the 128-byte fetches in the streaming benchmarks, the network bandwidth is only 1.2x larger than the cache bandwidth.

5.3.3 Latency

We have run simulations for 256 threads, 512 threads, and 1024 threads at 100 ns, 200 ns, 500 ns, 750 ns and 1000 ns read latencies with the write latency double that of the read latency to find the system's sensitivity to latency and parallelism. These results are shown in Figure 5.9. As expected, lower latencies increase ReRAM's performance for all the benchmarks at all thread counts. Reducing the access latency from 200 ns and 400 ns to 100 ns and 200 ns for reads and writes, increases the performance by 72% on average. Increasing the latency reduces per-

Table 5.3: Crossover Read Latencies (ns)

Benchmark	DRAM 4 Stacks			DRAM 8 Stacks		
	32B	64B	128B	32B	64B	128B
APSP	1187	1143	2188	569	560	1050
BC	664	641	882	344	329	417
PR	1337	1277	3161	530	505	1226
SSSP	1785	1826	3678	857	872	1688
Average	1243	1222	2477	575	567	1095
DAXPY	987	718	344	486	348	187
KM	536	469	110	509	452	91
NN	1032	326	259	974	252	178
PF	1223	945	371	590	473	174
Average	896	615	271	606	382	157

formance by 55%, 69%, and 77% on average for read latencies of 500 ns, 750 ns, and 1000 ns respectively.

This data allows us to calculate at what latency the ReRAM is on par with the HBM system. Table 5.3 shows the read latency (in nanoseconds) of the monolithic ReRAM memory system for each benchmark at 1024 threads where their performance equals that of the given HBM2 DRAM system. The HBM system is competitive against the monolithic system for streaming computations with 8 stacks at the current latencies, except for KM. KM is latency bound rather than bandwidth bound, so the latencies need to be much closer to DRAM to be competitive. The graph kernels show the system being able to tolerate read latencies over 1 μs, which is more than what we currently expect ReRAM to exhibit [64].

While the latency of writes doesn't directly impact the latency of the memory system from the cores' perspective, it increases the occupancy of the memory banks. If the memory banks are occupied by writes, the read requests, which do stall the

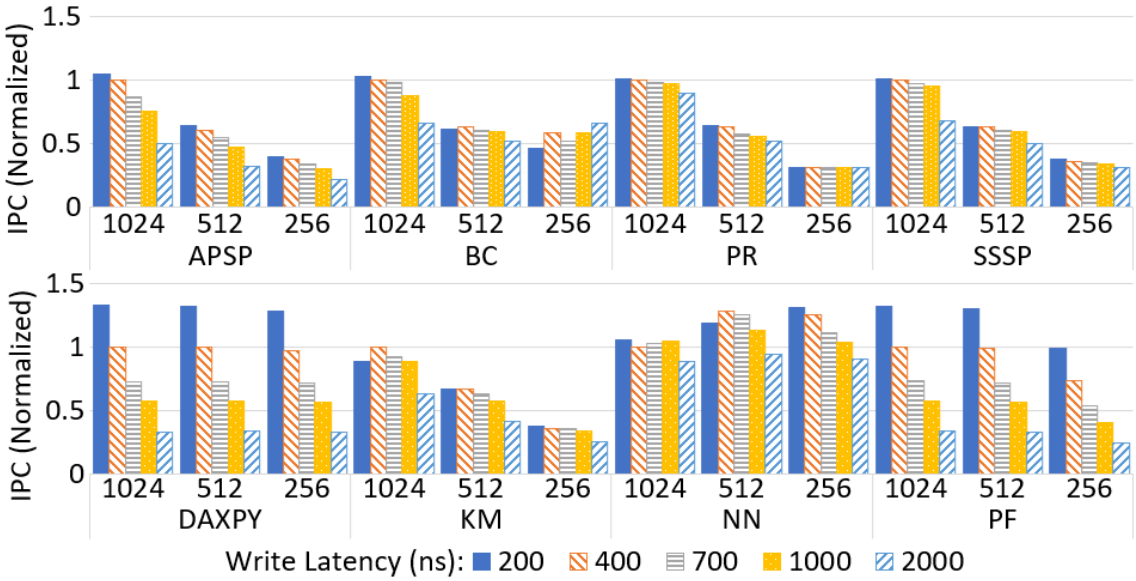


Figure 5.10: IPC of each benchmark as the write latency is varied for the monolithic ReRAM memory system, normalized to the 1024 threaded, 200 ns case.

cores, will experience an increase in effective latency due to queueing latency. While we expect write latency will be at least twice that of read latency, it could be even higher [64]. Reliability schemes that involve writing, checking and re-writing could reasonably increase the write latency to 5x that of reads. Additionally, there are schemes that increase the write latency to decrease energy or wear on the memory cells [28]. To examine the possible impact of these schemes on performance, we have swept the write latency independent of the read latency to determine how sensitive each benchmark is to higher write latency. We have run simulations for 256 threads, 512 threads, and 1024 threads with a read latency of 200 ns. In Figure 5.10, we have varied write latency from 100 ns to 2 us and report the performance normalized to the 400 ns, 1024 threads case.

As expected, the higher write latencies affect the benchmarks with the highest percentage of writes the most. DAXPY and Pathfinder both have about $\frac{1}{3}$ writes

and their performance degrades by 27% at 700 ns, 42% at 1 μs and 66% at 2 μs. At the very high write latencies, we begin to see it affecting all the benchmarks to a significant degree as the occupancy of the banks rises steeply. Excluding DAXPY and Pathfinder, the average decrease in performance is 4.5% for 700 ns, 4.6% for 1 μs, and 23% for 2 μs in the 1024 thread case. The impact of the 2 μs case is reduced to 18% for 512 threads, and 8.8% for the 256 thread case. PR has the smallest percentage of writes, only 3%, and so its performance only decreases by 11% at 2 μs and 1024 threads.

In BC with 256 threads, the cache behavior is very sensitive to configuration changes. The L2 cache miss rate changes from 30% to 9% as the write latency is increased. Reducing the rate at which new data is read from main memory increases the efficacy of the cache. This effect is present when increasing the read and write latency of memory requests, but is outweighed by benefits of the larger reduction in access latency. In Section 5.3.9, we will look at how changing the capacity of the cache affects the benchmarks which explains this effect in BC to some degree.

5.3.4 Banks

Another important design consideration is the number of ReRAM banks. We have run simulations for 256 threads, 512 threads, and 1024 threads with a read latency of 200 ns and write latency of 400 ns. We have varied the number of banks running simulations at the 32 banks/tile (8k banks total), 64 banks/tile (16k banks total), 128 banks/tile (32k banks total), 256 banks/tile (64k banks total), and 512

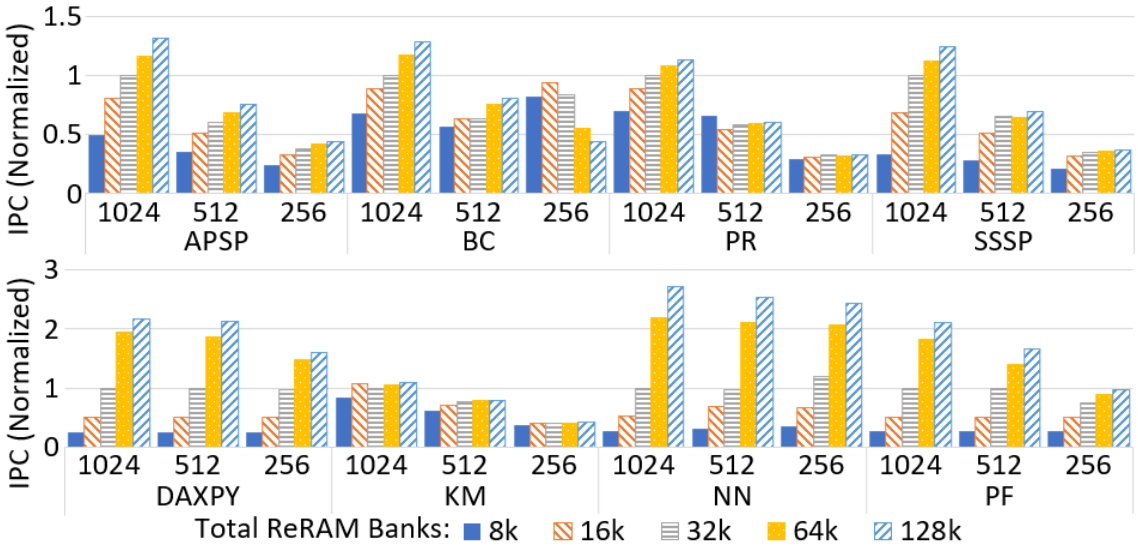


Figure 5.11: IPC of each benchmark as the total number of ReRAM banks is varied for the monolithic ReRAM memory system, normalized to the 1024 threaded, 32K banks case.

banks/tile (128k banks total) design points. Previously all our simulations were at the 128 banks/tiles (32k banks total) design point. These results are shown in Figure 5.11.

As expected, at higher thread counts, the number of banks has the largest impact. For the graph kernels, increasing the number of banks from 32k to 128k increases the performance by 24% at 1024 threads and 16% at 512 threads. At 256 threads, BC's cache sensitivity leads to a decrease in performance of 6%, but if BC is excluded, the performance increases by 8%. The increase in banks for the graph benchmarks is caused primarily by a decrease in bank conflicts.

In the case of the memory intensive streaming benchmarks (DAXPY, NN, and PF), the memory system is saturated by the memory systems requests, so increasing the banks increases the available bandwidth and performance. Additionally, since much of their parallelism is a result of prefetching rather than TLP, the performance

increase is still significant at lower thread counts. On average, increasing the number of banks from 32k to 64k results in a 99%, 82% and 48% increase in performance for 1024, 512, and 256 threads respectively. As you increase from 64k to 128k banks, the memory system is no longer saturated or the bottleneck, so increasing from 64k to 128k only results in a 17%, 18% and 11% increase in performance. NN derives the largest benefit, its performance increasing by 2.5x from 32k banks to 64k banks in the 1024 thread case, because increasing the number of banks quickly reduces the high contention caused by only heavily using a quarter of the banks due to its striding access pattern.

KM is compute bound so the number of banks only minimally affects the IPC, with an increase of 29% from 8k banks to 128k banks for 1024 threads.

5.3.5 Network Width

We have tried varying the size of the network channels for both the ReRAM and DRAM based systems. This gives us some insight into how dependent our performance is on the wide network created to not hinder DRAM performance.

We have run simulations for 1024 threads. The ReRAM system had 128 banks/tile and a read latency of 200 ns and write latency of 400 ns. The DRAM system is the 8-stack system with a 64-byte fetch. We have varied the width of the network channels running simulations at the 8-byte, 16-byte, 24-byte, 40-byte, and 72-byte design points. Previously all our simulations were at the 72-byte design point. The 72-byte design point allows a full DRAM packet (64 bytes of data plus

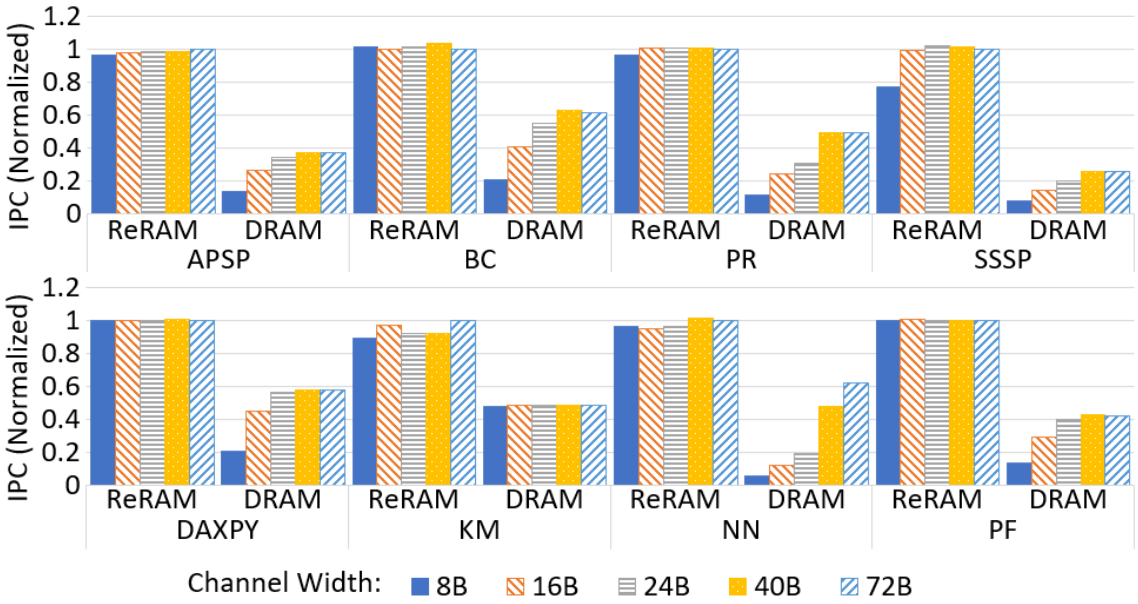


Figure 5.12: IPC of each benchmark as the network width are varied for the given memory system, normalized to the ReRAM 72-byte case.

an 8-byte header) to be transmitted in a single cycle.

Figure 5.12 shows most of the benchmarks experience little performance variation when varying the channel width. The most affected is SSSP. At 8 byte channels, its performance is only 80% of its peak, but, by 16 byte channels, SSSP has full performance. The DRAM systems do however have a noticeable difference below 40B channels. Excluding KM, which is not bandwidth bound, the average performance of the DRAM is reduced by 71%, 42%, and 22% for 8 bytes, 16 bytes, and 24 bytes respectively. While we could likely reduce the ReRAM channels to 8-16 bytes without harming performance, the DRAM system needs wide channels to carry all the data it produces.

One caveat is that the DRAM fetch width in these experiments is only 64B compared to the 128B fetch we often use for the streaming computations. If we did an additional sweep with the DRAM fetch width at 128B, we may see the

performance degradation happen at the 72B to 40B change.

5.3.6 Tiles

Another way to test changing the network is by reducing the total number of tiles. The cores are frequently idle with 256 cores even with 4 threads/core. By reducing the number of tiles to 64 in an 8×8 network, we can see the effect of a smaller network and fewer memory and cache controllers on the performance. We hold the number of threads and total number of banks constant for each of the configurations.

Figure 5.13 shows the performance of each benchmark. The different graphs represent the three thread counts—1024, 512 and 256. The different colored bars represent the sweep of total number of ReRAM banks from 8k to 128k total banks ReRAM banks. Finally the solid vs striped bars represent the 8×8 tiled design vs the 16×16 tiled design.

When we reduce the number of tiles while holding the number of total ReRAM banks and threads constant, we still get a reduction in performance as seen in Figure 5.13. This is due to contention in the network and at the memory-cache controllers. With the smallest number of total ReRAM banks, the ReRAM banks are the bottleneck of the system, so reducing the network and number of controllers by 4x has little impact on performance. As the number of banks is scaled up, the bottleneck shifts to the controllers and network. This effect is particularly pronounced for the streaming benchmarks which were using the peak of the resources

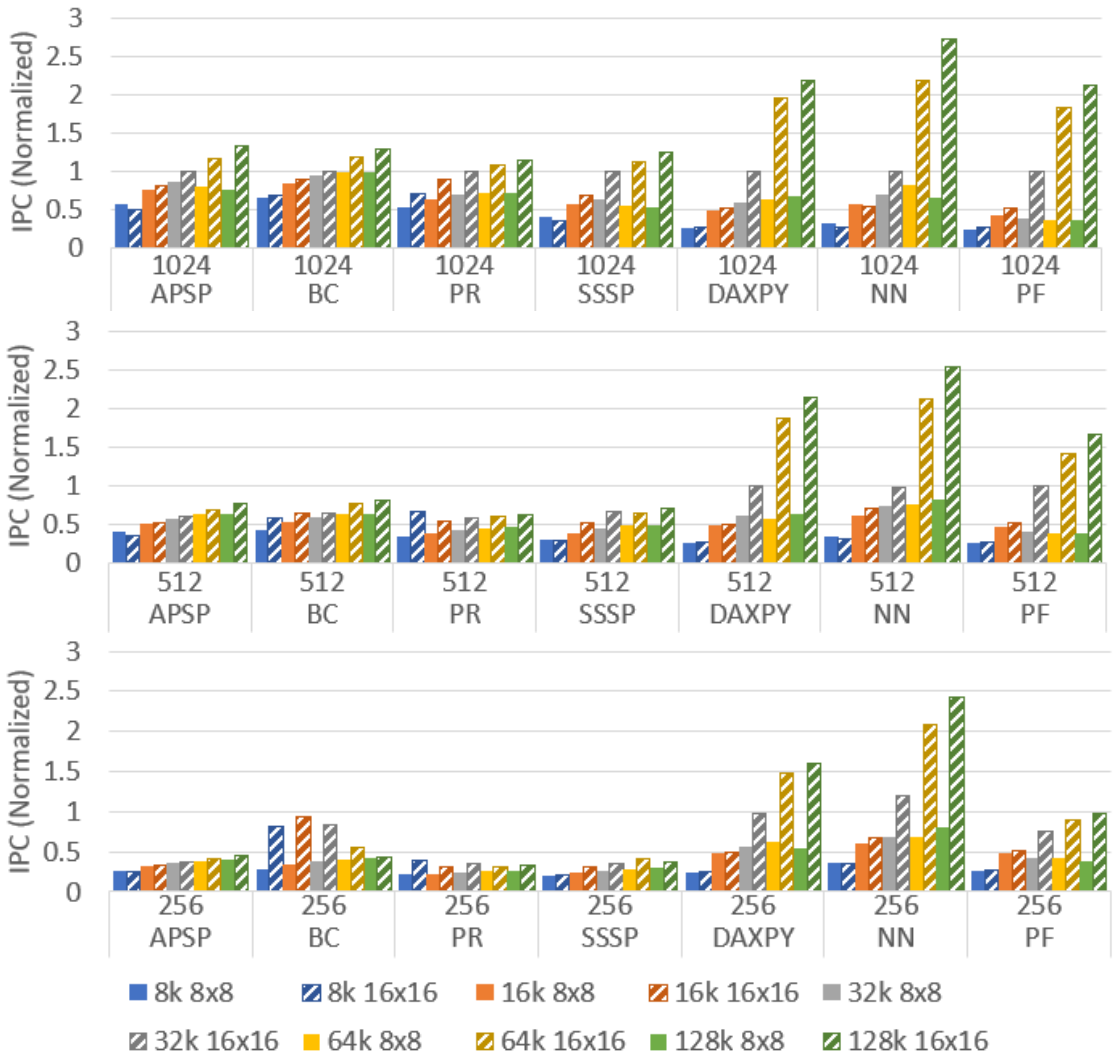


Figure 5.13: IPC of each benchmark as the number of tiles and total ReRAM banks are varied for the monolithic ReRAM memory system, normalized to the 16 by 16 tiles, 1024 threads, and 32K bank case.

in the 16×16 tile machine. At 1024 threads and 128k ReRAM banks, the streaming benchmarks' performance is reduced by an average of 76% compared to the graphs' performance which is reduced by 41%. The effect is also lessened to 13% for the graph benchmarks 256-thread case as much of the memory parallelism is TLP, whereas the streaming benchmarks still experience a 64% decrease in performance.

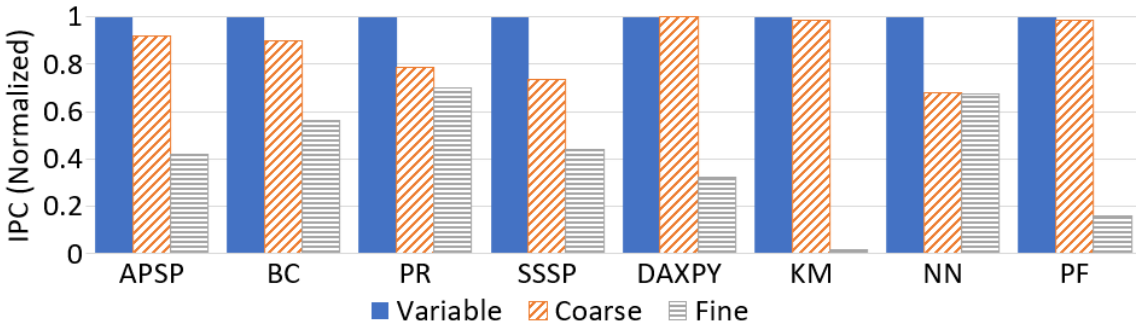


Figure 5.14: IPC of each benchmark with different fetch granularities for the ReRAM memory system, normalized to the variable fetch granularity case ("Variable").

5.3.7 Granularity

The monolithic system up to this point has been using a variable fetch granularity scheme. If fetching a single memory location or a vector load, an entire cache block is loaded. If the instruction is a gather, only 8 bytes will be fetched. This minimizes overfetch in the irregular benchmarks and NN which has a large stride while still allowing large accesses and the benefits of spatial locality in the cache. We have run simulations where the fetch is either only 64 bytes (coarse grain) or 8 bytes (fine grain) and graphed it in comparison to the variable fetch granularity in Figure 5.14 to see what benefit this scheme provides in terms of performance.

The graph benchmarks and NN, which make use of gather loads, suffer from only being allowed coarse grain fetches, their performance is reduced by 20% on average, with NN affected the most, decreasing by 32%. The bank occupancy rises, and there is more network contention when always fetching 64 bytes. The remaining streaming benchmarks, which never use gather loads, are unaffected by the change.

All benchmarks are negatively affected by only having fine grain fetches, on average their performance is reduced by 59%. Only performing fine grain fetches

makes prefetching far less effective. Eight prefetches per thread previously generated 512B of data and now only loads 64B of data. Even the graph kernels rely on prefetching edge arrays. It also removes any spatial locality benefits of the cache, dramatically reducing KM's performance to only 2% of its performance with coarse grain fetches. For a workload that is completely irregular, like GUPS, this would probably not affect performance, but for most real workloads, there will be benefits to coarse grain fetches—which is reflected in how memory systems are currently designed.

5.3.8 Co-Design and Logical Integration

All our results so far have assumed the co-designed L2-ReRAM main memory module. As mentioned in Chapter 4.3.1, without co-design, each tile would be 22.1% larger: instead of 256 cores, we would only have 210 cores in the same die area. Unfortunately, our simulator requires a power-of-two number of tiles, so we cannot evaluate 210 cores. To estimate the impact, we reduced each core's threads from 4 to 3 (25% reduction), and we reduced the number of per-tile ReRAM banks from 128 to 112 (14% reduction) to maintain a divisible-by-8 match with cache lines.

Figure 5.15 shows the performance of this reduced configuration, labeled "Monolithic-Separate-Design," normalized to the 1024-thread "Monolithic" bars from Figure 5.7. Figure 5.15 shows that on average, performance decreases by 18% for the reduced configuration. While this is not a precise experiment, it nevertheless shows our co-designed L2-main memory structure affords area efficiency that can directly translate

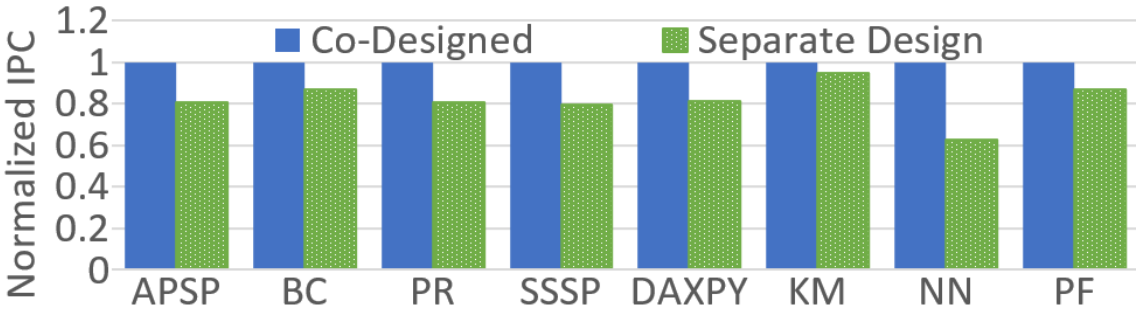


Figure 5.15: Co-designed versus a separately-designed system (approximated).

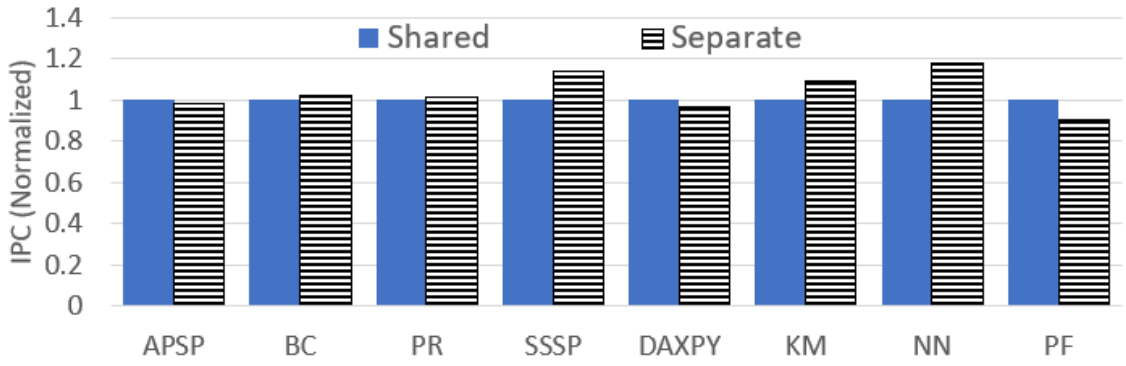


Figure 5.16: IPC of each benchmark with and without a interconnect exclusively for the ReRAM memory system, normalized to case with a shared interconnect ("Shared").

into performance gains.

Another design question is the impact of having a single shared interconnect to communicate with both the cache mats and with the ReRAM banks as compared to each system having its own interconnect. Up to this point, the simulations have had a shared interconnect configuration. We have simulated having separate interconnects to see what the impact is.

Figure 5.16 shows the two systems exhibit very similar performance, with an average of 4% better performance for the case with separate interconnects. As mentioned earlier, the single interface may increase contention due to the shared internal interconnect, which is why performance is better for the redundant system

in BC, PR, SSSP, KM, and NN. At the same time, the streamlined interface has lower latency, which is why performance degrades slightly for the redundant system in DAXPY and PF which are pushing the cache interface to its limit.

5.3.9 Cache Size

Our base configuration uses a 256 KB L2 cache slice, or 64 MB L2 cache. This was selected to be large enough to hold frequently and widely accessed data, such as base addresses, but small enough that most of the data operated on (e.g., graph nodes) would rarely be in cache for other threads. We are operating on smaller datasets than would be expected in real applications, so we aimed to have a smaller cache than what our cache slice designed in Chapter 3 dictated. However, we are still interested in what affects the cache may have if it is sufficiently large or smaller than the current point.

For the 16×16 tile machine, we have run simulations for 64 KB, 128 KB, 256 KB, 512 KB, and 1 MB cache slices, which translate to 16 MB, to 256 MB total L2 cache capacity. These results are presented in Figure 5.17. Additionally, we have run a similar sweep for the 8×8 tile machine, with cache slices from 64 KB to 4 MB, representing a total L2 cache of 4 MB to 256 MB. These results are presented in Figure 5.18.

As expected, for the graph benchmarks, increasing the size of the cache generally improves performance. The graph we are using for testing is a power law graph. This means that some nodes are very well connected and will be accessed

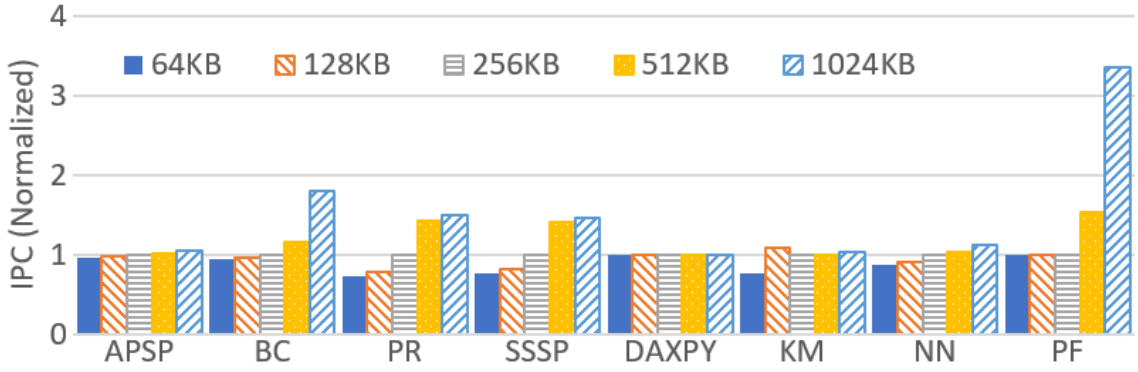


Figure 5.17: Sweep of L2 cache slice capacity for 16×16 tiles.

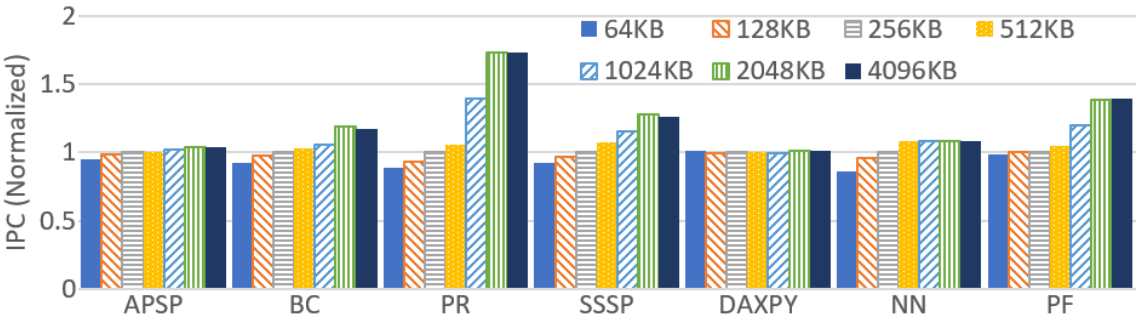


Figure 5.18: Sweep of L2 cache slice capacity for 8×8 tiles.

more often than others. The increase in capacity means a larger percentage of these nodes can be kept in cache leading to an increase in performance. For the 16×16 tiles, on average, the L2 hit rate increases from 41% in the 256 KB/tile case to 71% in the 1 MB/tile case resulting in a 45% increase in performance. With this layout, BC is most sensitive to the cache size, increasing in performance by 81% from 256 KB/tile to 1 MB/tile. For 8×8 tiles, the average increase from 256 KB/tile to 1 MB/tile is 16%. From 1 MB/tile to 4 MB/tile (which is the same total capacity as the 256 KB/tile to 1 MB/tile in the 16×16 tile case), performance increases by 13%.

Generally, the streaming benchmarks are less sensitive to cache capacity. These benchmarks have little sharing between threads. Take DAXPY for instance, which

calculates $y[i] = a * x[i] + y[i]$; each thread will always access a disjoint set of indices and the thread will not read any element multiple times. This leads to DAXPY being very cache-size independent. Pathfinder, on the other hand, will revisit previously accessed data, and so does benefit greatly from that remaining in the cache. There is a tipping point from the 32 MB or 64 MB L2 cache, where Pathfinder starts to see a large increase in performance due to a higher L2 hit rate; the L2 hit rate goes from nearly 0% to 80% increasing performance by 3.3x.

Another dimension to consider for cache organization is the number of logically independent cache banks. As discussed in Chapter 3, adding cache banks is one way to increase the parallelism of the cache, but may result in uneven bank utilization, decreasing performance. The two effects need to be balanced.

To simulate this, we swept the number of cache banks per cache slice from 1 bank to 8 banks for both tile sizes. Addresses were interleaved in 64-byte blocks to allow cache lines to stay together, but streaming accesses to stream across banks. The 32 MSHRs were divided evenly between all banks. If one cache bank exhausted its MSHRs, all banks would stall incoming requests as there is no re-ordering in the network queue.

Figure 5.19 shows the performance for the cache bank sweep for 16×16 tiles. In these cases, the number of cache slices, means there are 256 cache banks at the low end and 2,048 at the high end. With the pipelining scheme, 256 cache banks generally provide enough parallelism to achieve the best performance. Therefore, increasing the number of cache banks does not result in better performance. There is, however, a penalty from fewer MSHRs per bank. At 8 banks, there is a higher

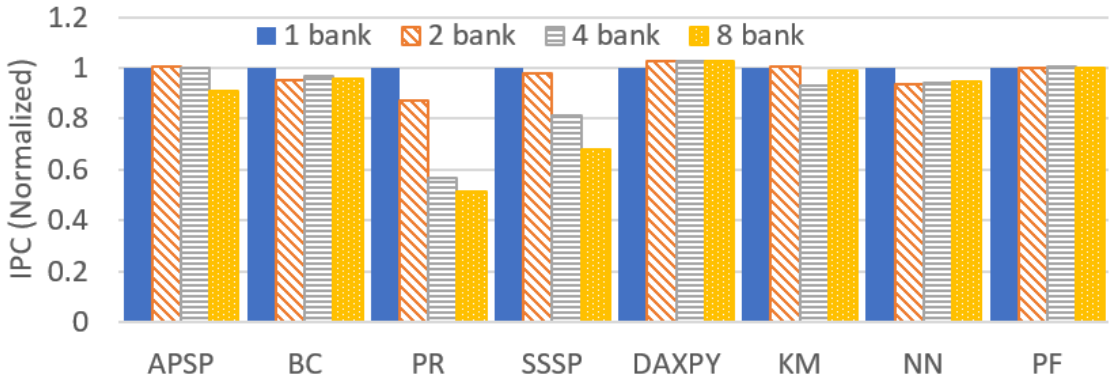


Figure 5.19: Sweep of L2 cache banks per tile for 16×16 tiles.

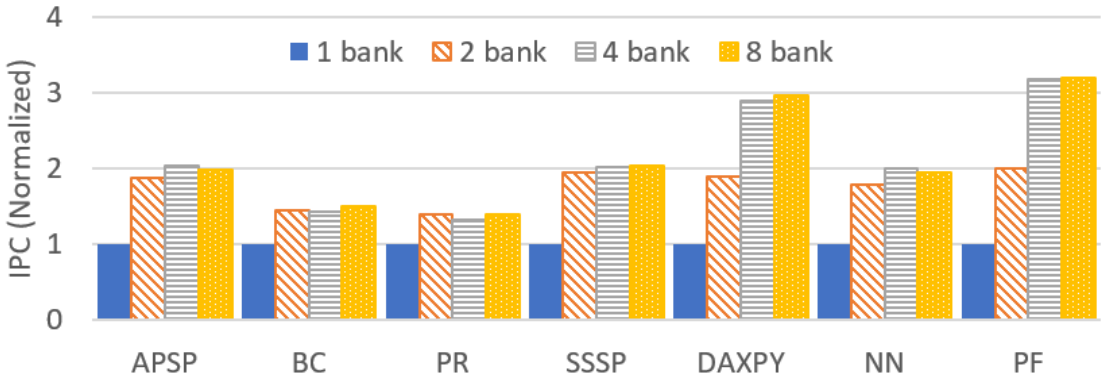


Figure 5.20: Sweep of cache banks per tile for 8×8 tiles.

likelihood that one particular bank will exhaust its 4 MSHRs, stalling the other cache banks on the same slice, than a single bank will exhaust all 32 of its MSHRs. This is evident mainly in the irregular accesses of the graph kernels, which are more likely to cause contention with each other, reducing performance by 24% on average. The streaming benchmarks have the benefit of evenly streaming through each of the cache banks due to the regular access pattern. This prevents a performance decline due to uneven utilization of resources.

Figure 5.20 shows the same result for 8×8 tiles. For these cases, there are only 64 cache banks at the low end and 512 cache banks at the high end. With only 64 cache banks, there is not enough parallelism in the cache to match the parallelism

from the other components (the cores and main memory). When increasing the banks in this scenario, the performance increases by 2.1x on average as the cache is able to satisfy more requests. This is particularly true for DAXPY and PF which use nearly the peak bandwidth of the main memory; their performance increases by 3.0x and 3.2x respectively.

5.3.10 MSHRs

Another consideration for cache design is the size of the MSHR file. As we saw in the cache bank sweep, a lack of MSHRs will negatively impact performance. However, having an overly large MSHR file requires a large amount of area and increases its access time.

The baseline configuration contains 128 ReRAM banks and so can service up to 128 outstanding requests simultaneously, potentially requiring 128 MSHRs. However, a few factors decrease the actual MSHR usage. The first is write requests do not occupy a MSHR. The second is that vector requests will occupy eight ReRAM banks but a single MSHR; if all requests were vector requests, only 16 outstanding requests could be serviced simultaneously. Even the irregular graph kernels make use of vector requests for the edge arrays. Finally, it is rare to occupy all banks in the benchmarks that do not exhibit a streaming pattern.

To determine how many MSHRs are sufficient for the best performance, we sweep the number of MSHRs per cache slice from 4 to 128. These results are presented in Figure 5.21. At 4 MSHRs the performance is reduced by 50%; at 8

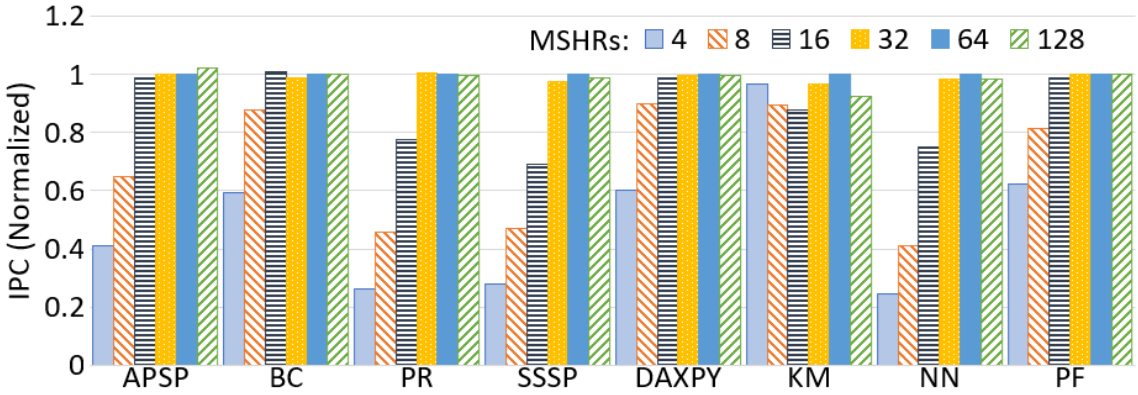


Figure 5.21: Sweep of L2 MSHRs per cache slice.

MSHRs performance is reduced by 32%. For most workloads, 16 MSHRs is sufficient to achieve the peak performance with 128 ReRAM banks. The exceptions are NN, which doesn't make use of vector loads, PR and SSSP, which seem to have more bank conflicts than the other graph kernels. All benchmarks are within 98% of peak performance at 32 MSHRs per tile.

5.3.11 Power

As discussed in Section 4.3.2, we do not have a definite number for ReRAM's access energy. Instead, we look at a variety of potential access energies and their impact on power. As a baseline, we assume 2.4 pJ/bit for ReRAM's read energy based on Crossbar's experience with their ReRAM technology [64]. We assume the best case for write energy, which is double that of read energy at 4.8 pJ/bit. DRAMSim3 provided the DRAM energy and for off-chip data movement to HBM2, we used 2.8 pJ/bit [94]. For data movement on the CPU chip, we assumed 0.1 pJ/bit/mm.

Figure 5.22 shows the power used in the memory system, including the LLC, network, data movement off the CPU chip, and RAM access (DRAM or ReRAM)

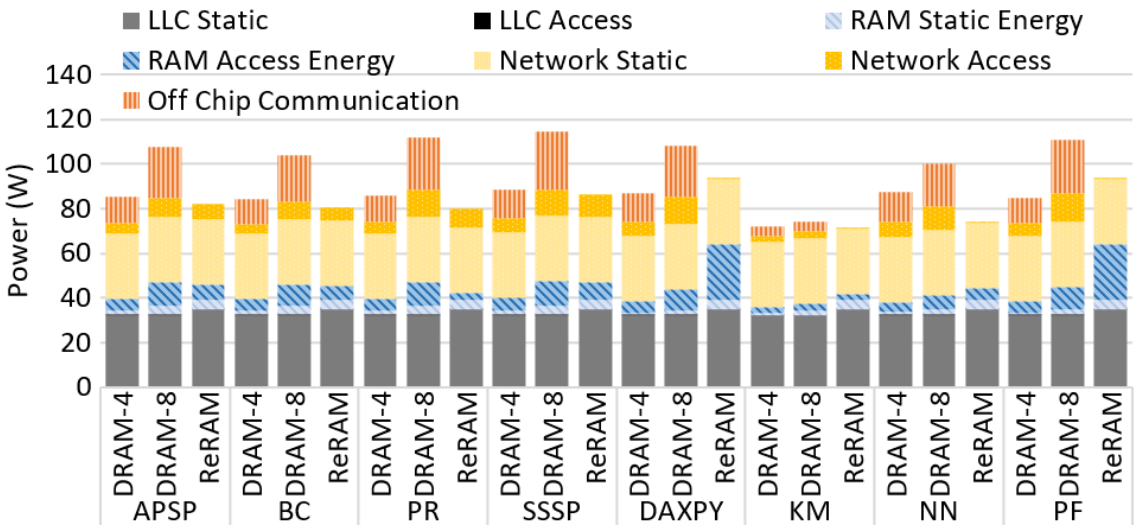


Figure 5.22: Power in the memory system.

for the 4-stack HBM2, 8-stack HBM2 and monolithic systems, labeled "DRAM-4", "DRAM-8" and "ReRAM," respectively. We only report results for the best-performing thread count per benchmark. Figure 5.23 shows the normalized energy in the memory system for all of the same components. Varying the thread count affects the dynamic energy by a few percent as roughly the same total amount of work is performed; however, static energy grows with execution time.

As Figure 5.23 shows, the monolithic memory system consumes less energy than HBM2. On average, for the graph kernels the monolithic system uses 4.6x and 3.0x less energy compared to 4- and 8-stack HBM2 respectively and for the streaming computations the monolithic system uses 1.2x less energy compared to 4-stack HBM2 and nearly the same, 0.98x, as 8-stack HBM2. The monolithic system does not incur any off-chip data movement. For the streaming computations, the monolithic system eliminates most of the on-chip data movement as well since memory accesses are almost entirely localized within each tile. Additionally, the graph

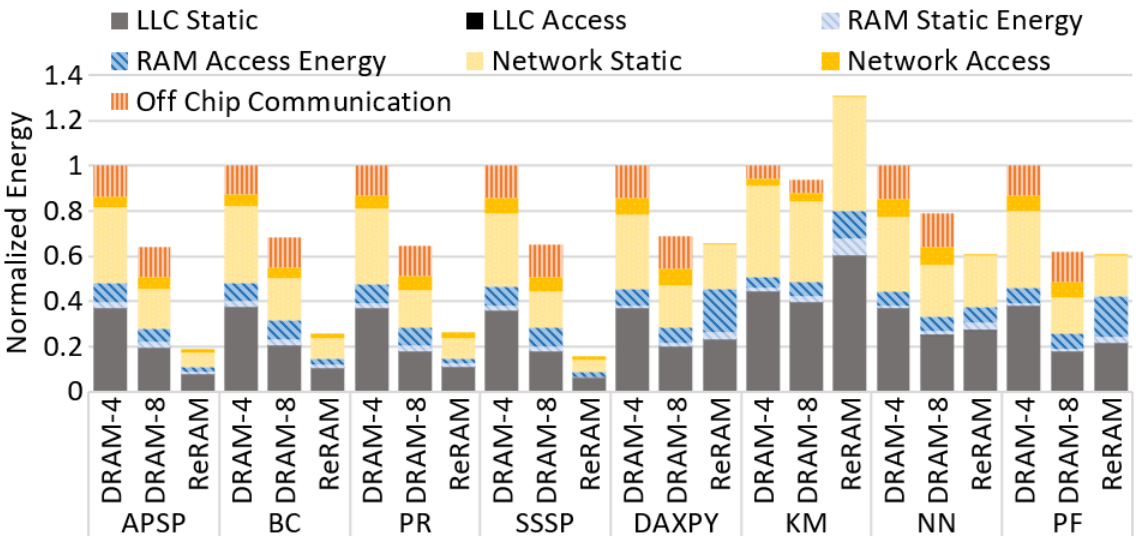


Figure 5.23: Energy in the memory system.

kernels and NN exhibit sparse memory access patterns for which HBM2 incurs overfetch. These benchmarks roughly use $\frac{1}{4}$ of the data fetched from DRAM during scatter-gather accesses (8 bytes out of every 32 byte fetch for the graph kernels and 32 bytes out of every 128 byte fetch for NN). This increases all components of HBM2 energy consumption relative to the monolithic system in these benchmarks. Finally, the improved performance and lower execution time means that the ReRAM system consumes less static energy.

The write energy has the largest amount of uncertainty. Writes are asymmetric, with SET operations incurring 10x the energy cost of RESET operations. Additionally, since ReRAM currently is proposed as a storage class memory, written data needs to last on the order of years. Writing at this intensity could require 120 pJ/bit. However, when using ReRAM for main memory, the retention requirement can be relaxed, resulting in less energy intensive writes. These soft writes could reduce the write energy to 4.8 pJ/bit. Due to this uncertainty, we sweep the

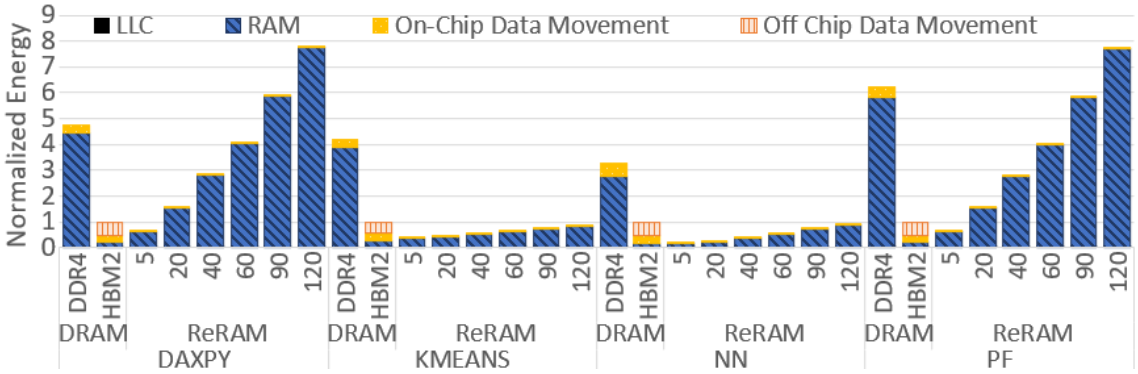


Figure 5.24: Dynamic energy in the memory system for streaming benchmarks.

required write energy from 5 pJ/bit to 120 pJ/bit. We also include energy results from DRAMSim3 for a discrete system that uses 64 GB of DDR4 DRAM and the 8 stack HBM2 DRAM system.

The results for the streaming benchmarks are presented in Figure 5.24. DAXPY and PF are the most write intensive benchmarks used, both have about $\frac{1}{3}$ write accesses. This becomes particularly apparent as we increase the write energy. While they remain competitive with DDR4 memories up to the 60 pJ/bit, they consume more energy than HBM2 much sooner. It is likely that if typical workloads are write intensive, the system they run on will require schemes to minimize the number of SET operations and possibly use DRAM as a write buffer to minimize writes to the ReRAM memory.

The results for the graph benchmarks are presented in Figure 5.25. None of the graph benchmarks selected are write intensive, so even increasing the energy of writes to over 100 pJ/bit has little effect on their dynamic energy, especially when compared to the energy required by DDR4. If the typical workloads of a system are reading over 90% of the time, a ReRAM only system may be preferable.

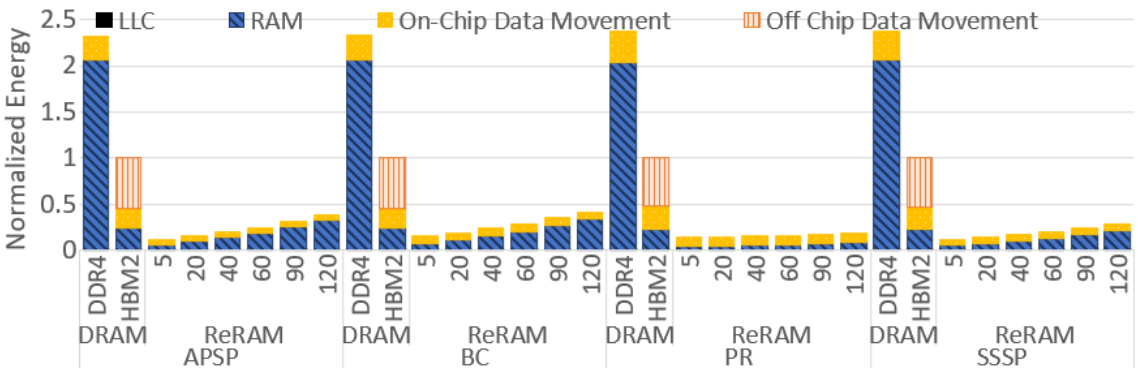


Figure 5.25: Dynamic energy in the memory system for graph benchmarks.

Table 5.4: System Analytical Model Parameters

Component	Use
Miranda	Pattern-based CPU model
Merlin	Network
MemHierarchy	Cache and main memory
Messier	NVM memory model
DRAMSim3	DRAM memory model

5.4 SST Comparison

We compare our results to a validated simulator, Sandia National Laboratories' Structural Simulation Toolkit (SST) [95]. Previous work has used this simulator for architectural simulations of a monolithic ReRAM system [79]. We model our experiments after this work, using the same components, which are listed in Table 5.4.

For the pattern-based CPU model, we use the STREAM pattern, which has the same memory access pattern as DAXPY ($z[i] = a * x[i] + y[i]$), and GUPS which reads then writes pseudo random locations in memory. To account for the vectorization present in our benchmarks, the STREAM benchmark used 64-byte operands and GUPS had an 8 deep re-order buffer to allow 8 outstanding read

requests similar to the 8 outstanding gathers. We limit the simulation to one thread per core.

We did not sector the cache; instead, we only used coarse-grained requests within our simulator. The L1 cache was private with a capacity of 32 KB and 4-way associativity. It used the Cassini stride prefetcher with a reach of 32 to match our simulator. The L2 cache was distributed and shared, with 256 KB per slice and 8-way associativity. The memory model we used in SST is coherent, but the coherence traffic was minimal and should not meaningfully affect the performance.

We limited the network channels to 8-bytes and the number of tiles to the 8×8 tile configuration. The SST configuration uses a single mesh network and SST handles any deadlock issues. Our simulator uses a single mesh network and virtual channels to separate traffic from cache to CPU, DRAM memory controllers to cache, cache to DRAM memory controllers, and finally, CPU to cache with priority in that order to prevent any deadlocks.

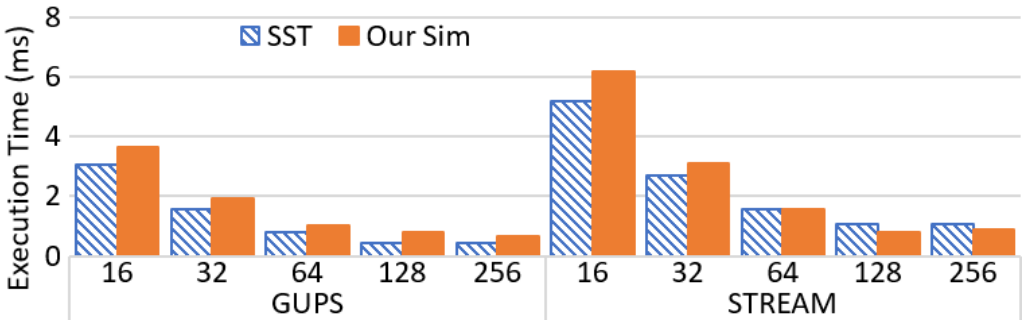
The Messier model allows us to set asymmetric read and write latencies. Since we switched to only coarse-grained requests, we model $\frac{1}{8}$ the number of banks with a request width of 64 bytes instead of 8 bytes. Modeling an increasing number of banks in the Messier model did not change the execution time of the benchmarks. However, increasing the number of controllers per tile does and so we model multiple ReRAM banks per tile with controllers. This introduces some inaccuracy as there is more parallelism in the controller, but since the network should only deliver a single request per cycle, the difference should not cause a large performance difference.

To model the HBM2 DRAM memory system, both simulators use DRAMSim3.

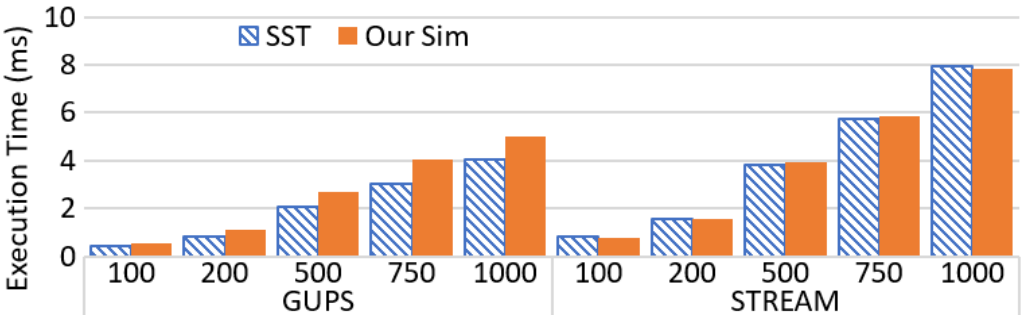
We use a single channel per memory controller, with memory controllers placed on the tiles on the edge. For the SST simulator, the DRAMSim3 configuration file only has a single channel which is duplicated in each memory controller. For our simulator, the DRAMSim3 configuration file contains all channels, and the simulator places the channels where needed.

We performed 3 experiments per benchmark. These compare the impact of increasing the number of ReRAM banks, of changing the ReRAM access latency, and comparing against a HBM2 DRAM system. Figure 5.26a shows the execution time when sweeping the number of ReRAM banks from 16 banks/tile (2 banks/tile in the SST simulator) to 256 banks/tile (32 banks/tile in the SST simulator) with a 200 ns read latency and 400 ns write latency. Figure 5.26b shows the execution time when sweeping the ReRAM read latency from 100 ns to 1000 ns with write latency double that of reads and 64 banks/tile (8 banks/tile in the SST simulator). Figure 5.26c compares the execution time of ReRAM with 200 ns read latency, 400 ns write latency, and 64 banks/tile (8 banks/tile in the SST simulator) against the execution time of HBM2 DRAM with 8 channels/stack, a 64-byte fetch width and one or two stacks labeled "DRAM-1" and "DRAM-2", respectively.

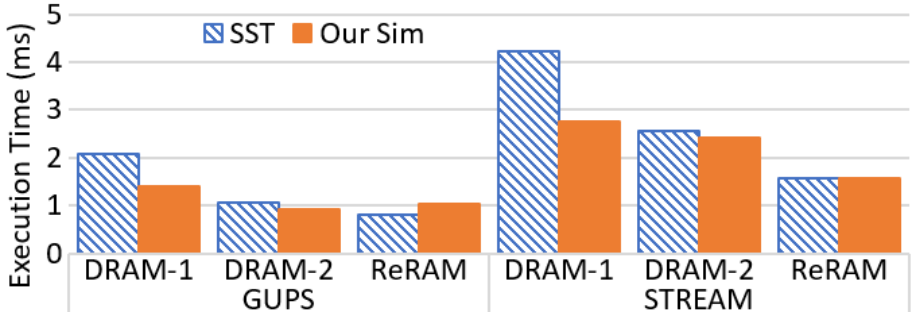
Our simulator agrees fairly well with SST. For the STREAM benchmark, the average percent error was 10%; for GUPS, the average percent error was 24%. For GUPS, our simulator consistently underestimated the ReRAM performance and overestimated the DRAM performance compared to SST, which may mean the benefit we show for monolithic performance versus the HBM2 DRAM system is a conservative estimate for the irregular benchmarks. Beyond the error in nominal values,



(a) SST vs our simulator bank sweep.



(b) SST vs our simulator latency sweep.



(c) SST vs our simulator dram sweep.

Figure 5.26: SST vs our simulator

the trend for the GUPS sweeps in our simulator closely followed the trend seen in SST. For STREAM, the number of banks in our simulator has a larger impact on performance than in SST. This may be due to different overlapping of the streams to memory banks. The latency sweep was very accurate for STREAM, with only a 2% average percent error. While in STREAM our simulator often overestimates the performance of both the monolithic and HBM2 DRAM memory systems, it is a larger overestimation in the case of the HBM2 memory system, so our estima-

tion of the benefits of the monolithic system for streaming computations may lean conservative.

5.5 Conclusion

Based on the target architecture in Chapter 4, we developed a Pin-based many-core simulator that accurately models the memory behavior. Irregular sparse access graph kernels and dense, regular streaming computations were selected, parallelized, vectorized, and in the case of the streaming computations, localized to achieve high parallelism on the simulator.

We show that a monolithic main memory system outperforms a state-of-the-art HBM2 DRAM system for these benchmarks on this manycore architecture. We look at varying many of the components of the architecture, including number of tiles, network width, and the size of the cache. We particularly vary ReRAM parameters like access latency, energy, and number of banks to account for the continuing development of ReRAM as a memory technology. We also show the benefits of the variable granularity memory system, of the area saved by the co-design, and the minor cost of the streamlined interface.

Finally, we compare against a validated simulator to show our results are reasonable. This comparison also shows we may be underestimating the benefits of the monolithic memory system as our simulator consistently showed better performance for the HBM2 DRAM simulations than the SST simulator.

Chapter 6: Analytic Model

While we have run many simulations, they only map a fraction of the wider design space and only for a limited number of benchmarks. Each simulation data point takes days to generate, and, as we expand the parameters we are looking at, it is beneficial to have a lighter weight model giving some insight into what performance could be expected for a wider variety of systems. To accomplish this, we have created an analytic model for our monolithic tiled system. This analytic model is used to pinpoint the important factors to achieve performance and find architectural balance for the system explored in our simulations.

6.1 Background

An architecture is said to be balanced if the computation time is equal to the I/O time [96]. In other words, architectural balance occurs when all components of a system are fully utilized with no idle time due to waiting on another component in the system. If the cores must wait idly for requests to the memory to complete, the memory is under-provisioned with regards to the compute system; if the memory system is idle, it is over-provisioned with regards to the compute system. We can analytically model the components of the system to find their capacity for work and

resource usage, then balance them to each other to find good architectural balance.

When looking at analytic models for parallel CPUs and memory systems, there are many existing models for things like the network-on-chip, cache performance, tiled systems, parallel systems, and bank contention. However, we found none which combine all these elements. The closest ones primarily focus on the parallelism present in the compute compared to the parallelism in the memory. Part of this, we believe, is due to most shared memory machines not being able to saturate the network-on-chip given the blocking nature of memory references. Our system has much more parallelism and many more messages being shared than is typical of cores solely communicating with the memory system. While there is no exact match, we can use the ideas present in the literature to inform the development of our own model.

Many of the analytic models described in the literature focus on complexity and performance analysis of algorithms and are often based on the RAM model. These include Parallel RAM (PRAM) [97], Block-Transfer model (BT) [98], Hierarchical Memory Model (HMM) [99], the Uniform Memory Hierarchy model (UMH) [100] and LogP [101]. These give a fairly high-level view of the systems they model and do not allow much change in the underlying machine assumed. They do however highlight important parameters for algorithm performance.

There is a detailed analysis of mesh networks that can help guide any network components of our system [1]. Additionally, there is analysis of streams and the types of conflicts created that can be applied to the streaming half of our benchmarks [102, 103]. The graph kernels have mostly random traffic patterns and so can be handled

well probabilistically.

Bank conflicts and memory system models [104, 105] generally model the memory system as a queue with a probability of conflicts and queueing time. These models try to find the typical distributions of when a request will arrive at a particular memory queue and what the service time is for each request.

There has also been work done on the performance of GPUs which focuses on large numbers of cores and threads, and throughput computing rather than latency bound computing [106, 107, 108, 109]. Though our manycore architecture is still more latency bound than GPUs, it adopts similar mechanisms of throughput driven computing with a high number of threads and memory prefetching. These GPU models can give insight into handling the unique issues of a very parallel system but generally treat the memory system as fixed latency with a maximum load.

An analytical model that tries to account for many types of parallelism and includes many of the components of interest but in a more generalized way than the GPU models can be found in the X-model [110]. This model tries to find a balance between parallelism in the compute system and the memory system and includes complex cache effects. The model, however, does not really model queuing latency in the memory system nor network limitations.

6.2 Model

Our system can be thought of as consisting of four individual components (the ReRAM, the cache, the network, and the CPU) that interact with each other. In a

well-balanced system, the parallelism of the individual components will be able to sustain the parallelism afforded by each of the others. Our model works out how many requests each of the components can handle per second and how the interaction between the components in the system affects the total number of requests the CPU can send per cycle.

At the highest level, the number of parallel requests the system can sustain, P_S, is based on the smallest parallelism from the CPU (P_R), the main memory (P_M), the cache (P_C), or the network (P_M) as described in Equation 6.1. The model can then use the size of the requests, b_r and the cycle time, c to calculate average bandwidth of the system, B_S as shown in Equation 6.2.

$$P_S = \min(P_R, P_M, P_C, P_N) \tag{6.1}$$

$$B_S = P_S \times b_r \times c \tag{6.2}$$

Table 6.1 summarizes the inputs and outputs of the high level model. The next sections will go into detail about how to calculate the request parallelism of each of the subsystems and will include their individual model parameters.

6.2.1 CPU Parallelism

The CPU parallelism can be thought of in a few distinct ways. The first is how many requests can be sent to the memory system if it tried to constantly send requests, R_m. The second is how many requests does the workload want to send,

Table 6.1: System Analytical Model Parameters

Variable	Meaning	Unit	In/Out
P_S	System Parallelism	Requests / cycle	Output
P_R	Request Parallelism	Requests / cycle	Output
P_M	Main Memory Parallelism	Requests / cycle	Output
P_C	Last Level Cache Parallelism	Requests / cycle	Output
B_S	System Bandwidth	Bytes/s	Output
b_r	Bytes per request	Bytes / request	Input
c	Clock Speed	Hz	Input

R_p; if the program is very compute intensive, it may send few requests, even in a very parallel system. The final is how often requests are returned so it can send new requests, R_r. Equation 6.3 says we take the minimum of these values and that is the CPU's request parallelism, P_R.

$$P_R = \min(R_m, R_p, R_r) \tag{6.3}$$

Equation 6.4 describes the maximum number of requests the CPU can make to the memory system in a single cycle. The CPUs we are modeling include SIMD operations and variable granularity. To capture the different sizes of the possible requests, we have opted to treat the minimum width request as a single memory request, and any larger requests as multiple memory requests within the model. This means in the system we have simulated in Chapter 5, an 8-byte request would be 1 memory request and a 64-byte request would be 8 memory requests. The average vector size, v, is how many memory requests, on average, a CPU request contains. N is the number of tiles within the system. Each tile can send a single packet per

cycle, which will contain v memory requests.

$$R_m = vN \tag{6.4}$$

The request rate of the program is described in Equation 6.5. We use misses per 1,000 instructions (MPKI) as our measure of how memory intensive a program is. This metric counts the number of L1 cache misses that occur when executing 1,000 instructions. For scatter-gather requests, each individual memory request can miss, thus it is possible to have an MPKI greater than 1,000 for our system. However, if a program is very compute intensive and has a good L1 hit rate, the MPKI can reasonably be less than one. Since we are assuming a 1 IPC machine, we can divide MPKI by 1,000 to find how many misses we expect per cycle from the benchmark. This is then multiplied by the average number of memory requests per miss, v, and the number of CPUs, N, to find how many memory requests we expect the program to send per cycle, R_p.

$$R_p = \frac{\text{MPKI}}{1000} vN \tag{6.5}$$

Finally, there is how often we receive a reply to a request. For a given CPU and benchmark, there can only be so many outstanding requests at any one time. At a certain point, the thread will stall, the CPU will not have other threads to switch to, and the prefetcher will run out of prefetch buffers. The only event that will trigger new memory requests is receiving a reply to a previous one so that a thread can resume execution. The rate at which this happens, R_r, is based on the

number of outstanding requests that can be sustained, o_R, and the average latency for a request, L_A, as seen in Equation 6.6.

$$R_r = \frac{o_R}{L_A} \tag{6.6}$$

The number of sustainable outstanding requests is the blocking request plus all the non-blocking requests that happen between blocking requests multiplied by the number of threads, t, as shown in Equation 6.7. There are multiple sources of non-blocking memory requests in our model: prefetching, write requests, and nonblocking reads. We have simulated a system with prefetching and write requests, but not explicit nonblocking reads or any out of order execution that would result in some nonblocking reads. However, they are still a source of memory parallelism that can be generated by the cores.

$$o_R = (v_r + \min(\frac{f}{t}, s)v_p + (1-\omega)v_w + \gamma v_n)t \tag{6.7}$$

In our system, blocking requests are in the form of reads originating from the program (as opposed to the prefetcher). Since we have a SIMD machine, the blocking read instruction may contain several read requests; if it is a vector read, it will contain 8; if it is a gather, it can contain from 1 to 8. Since the thread does not stall until it sends all requests from the blocking instruction, we can have multiple outstanding blocking requests. This is captured by v_r, the average number of read requests per blocking read instruction.

Memory requests due to prefetching are captured in the second term in Equa-

tion 6.7. The prefetch depth, f, denotes the maximum number of requests the prefetcher can have in flight, shared among all threads. The number of outstanding requests a single thread can have in flight, which is dependent on the workload, is s. If a workload only uses data the prefetcher is prefetching, s is infinite, and the number of prefetches will be bounded only by the architecture. Essentially the prefetcher can never outpace the program, as every time it receives data, the data is consumed, and a new request is sent. The maximum number of requests the hardware will support are in flight at all times. If the benchmark consumes one piece of prefetched data for every non-prefetched read request, s will be one. The prefetcher can match the pace of the program and send a new request as the previous one is consumed, which should happen about as often as the prefetcher's requests are satisfied. If multiple pieces of data need to be requested per every piece that is prefetched, the prefetcher will outpace the program, and have no requests in flight for portions of the execution, making s less than one. Either s or f will limit the number of requests made. The average number of memory requests per prefetcher request is v_p; this is generally 8 in our evaluated architecture but could be changed.

In our architecture, writes are non-blocking. When a write occurs, the CPU sends that data to cache and continues with execution. The writes are represented by the third term in Equation 6.7. The percentage of reads is ω and the percentage of writes is $1-\omega$. This is, similarly to other terms, multiplied by the average width of write requests in the system, v_w. If it is a vector write request, it will have 8 memory requests, and scatter requests can have 1-8 requests.

The final type of potential non-blocking requests modeled in Equation 6.7

is non-blocking reads. While we don't implement these in our simulation model, it is another way of generating memory parallelism from the compute side. To calculate additional outstanding requests added by non-blocking requests, we need the percentage of nonblocking reads, γ and the average width of their memory requests, v_n.

The other important term in Equation 6.6 is the average latency of reply packets. L_A is the term which captures the majority of the interactions between the systems. The high-level equation is shown in Equation 6.8. Each of the terms is the latency of the subsystem with contention. The network latency is doubled to account for the two network traversals required of read packets. The main memory latency, L_{MC} is reduced by the L2 miss rate, m_{L2}, as only requests that miss in the L2 cache will have the additional main memory latency. Equation 6.8 hides a lot of complexity; each of the latency terms is affected by P_S which is in turn affected by L_A. The equation is solved using a numerical solver to find the balance point.

$$L_A = 2L_{NC} + L_{CC} + L_{MC}m_{L2} \tag{6.8}$$

Table 6.2 summarizes all the parameters and outputs of the core's model. The next sections will give the details for how the latency with contention for the other subsystems is calculated.

Table 6.2: CPU Analytical Model Parameters

Variable	**Meaning**	**Unit**	**In/Out**
P_R	Request Parallelism	requests/cycle	Output
R_m	Maximum Memory Requests	requests/cycle	Output
R_p	Maximum Program Requests	requests/cycle	Output
R_r	Maximum Returned Requests	requests/cycle	Output
N	Nodes in the system	tiles	Input
MPKI	Misses per 1,000 Instructions	-	Input
o_R	Sustainable Outstanding Requests	requests	Output
L_A	Average Reply Latency	cycles	Output
v	Average Request Width	-	Input
v_r	Blocking Read Width	-	Input
v_p	Prefetch Width	-	Input
v_w	Write Width	-	Input
v_n	Non-blocking Read Width	-	Input
t	Threads	-	Input
f	Prefetch depth	-	Input
s	Active prefetch streams	-	Input
ω	Read percentage	%	Input
γ	Nonblocking read percentage	%	Input
L_{NC}	Network Latency with contention	cycles	Output
L_{CC}	Cache Latency with contention	cycles	Output
L_{MC}	Memory Latency with contention	cycles	Output
m_{L2}	L2 miss rate	%	Input

6.2.2 NoC Parallelism

The network model is primarily based on Agarwal's detailed analysis of mesh networks [1]. Our model currently assumes a mesh network with separate channels in both directions and no end-around connects. Agarwal's model considers other topologies, but we only present the equations for our target architecture in this section.

Mesh networks are characterized by their dimensions, n, and number of nodes in a direction, k, with a total of $N = k^n$ nodes. Messages on average will take k_d hops in each direction to reach their destination and will have B flits per message. (In this work flits and phits are equal and are equal to the number of bits transferred over a channel in a clock cycle.) For random traffic, the average hops per dimension per message is given by Equation 6.9.

$$k_d = \frac{k - \frac{1}{k}}{3} \tag{6.9}$$

The probability of a packet arriving at an incoming channel, ρ, equivalent to the channel utilization, is given by Equation 6.10. This is the probability of a network request on any given cycle from a processor, m, multiplied by the number of flits per message, B, and the hops each message takes, nk_d, distributed to each of the channels associated with a node, $2n$.

$$\rho = \frac{mBk_d}{2} \tag{6.10}$$

The average contention delay through a switch, w, is given by Equation 6.11. This is used to calculate the average delay of a packet. Each hop of the packet will encounter $1 + w$ cycles of delay, plus the time for each flit to arrive sequentially, B. The final latency for a message with contention, L_{NC}, is given in Equation 6.12.

$$w = (\frac{\rho B}{1-\rho})(\frac{k_d - 1}{k_d^2})(1 + \frac{1}{n}) \tag{6.11}$$

$$L_{NC} = (1 + w)nk_d + B \tag{6.12}$$

Within our model, we would like to balance the number of packets sent and the amount of latency experienced by those messages. In other words, we would like to maximize the expression Equation 6.13 to find the value of m.

$$\frac{Nm}{L_{NC}} \tag{6.13}$$

When maximizing, we must keep in mind that both ρ and m are probabilities which must fall into the range $(0, 1)$. Since we are taking m as the dependent variable, we find the range for m which would also satisfy the range for ρ (Equ. 6.14) and restrict m to that range or $(0, 1)$, whichever is smaller (Equ. 6.15).

$$0 < \rho < 1$$

$$0 < \frac{mBk_d}{2} < 1 \tag{6.14}$$

$$0 < m < \frac{2}{Bk_d}$$

$$0 < m < \min(\frac{2}{Bk_d}, 1) \tag{6.15}$$

With our bounds established, we can then find the value of m which gives the maximum number of requests per latency. This is expressed in Equation 6.16. (A method not involving *argmax* would be to take the derivative with respect to m, set the derivative to 0 and solve for m.) This will give the ideal value for the probability a message is sent by a core; to find the expected parallelism of the network, P_N, we simply multiply by the number of cores. Finally, we need to include the factor for how many memory requests a single message can contain, v, due to our variable granularity requests.

$$m = \underset{m=(0,\frac{2}{Bk_d})}{\arg\max} \frac{Nm}{L_{NC}} \tag{6.16}$$

$$P_N = vNm \tag{6.17}$$

When calculating the total average latency, we use P_S to find the expected value of m. Since there are limits on the range of m, we similarly need to place limits on P_S to ensure m and ρ fall within the acceptable range for probabilities.

Table 6.3: Network Analytical Model Parameters. Source: [1]

Variable	**Meaning**	**Unit**	**In/Out**
P_S	System Parallelism	Requests / cycle	Output
P_N	Network Parallelism	Requests / cycle	Output
k_d	Average Hops	-	In/out
n	Network Dimensions	-	Input
k	Nodes/dimension	-	Input
N	Total Nodes (Tiles)	-	Output
B	Flits/message	-	Input
ρ	Channel Utilization	%	Output
m	P[Core sends request]	%	Output
w	Switch Contention Delay	cycles	Output
L_{NC}	Network Latency with Contention	cycles	Output
v	Average Request Width	-	Input

The case where m is 1 describes the maximum number of requests that can be sent by a core, R_m, which we take into account in the CPU parallelism model. So, we only need to add a constraint to P_S when m is instead constrained by the value of ρ. Equation 6.18 gives the range acceptable for P_S to ensure that the channel utilization of the network, ρ, remains under 100%.

$$\begin{aligned} 0 < m &< \frac{2}{Bk_d} \\ 0 < \frac{P_S}{vN} &< \frac{2}{Bk_d} \\ 0 < P_S &< \frac{2vN}{Bk_d} \end{aligned} \tag{6.18}$$

Table 6.3 summarizes all the parameters and outputs of the network-on-chip's model.

6.2.3 ReRAM Parallelism

The main factors that affect the maximum sustained bandwidth of the ReRAM main memory system are the number of banks and the access latency. Equation 6.19 is the basic equation for the number of requests the ReRAM can service per second, where P_R is the parallelism of the main memory, b_M is the total number of main memory banks, and L_M is the average latency of the main memory.

$$P_R = \frac{b_M}{L_M} \tag{6.19}$$

Since ReRAM has asymmetric access times, the proportion of reads to writes in a benchmark partially determines the average memory latency. The average latency is described by Equation 6.20, where ω is the percentage of reads, L_r is the read latency, and L_w is the write latency.

$$L_M = \omega L_r + (1 - \omega) L_w \tag{6.20}$$

While we have described the maximum parallelism from the ReRAM, often benchmarks will have bank conflicts, resulting in uneven utilization of the banks. If not all banks are in use at all points, the maximum parallelism will be less than we've predicted. We can add a conflict term to represent this loss. Equation 6.21 includes this conflict term, c_M, which represents percent of time a request goes to an occupied bank while open banks exist.

$$P_R = \frac{(1 - c_M)b_M}{\omega L_r + (1 - \omega)L_w} \tag{6.21}$$

The number of conflicts is dependent on the benchmark. If there are any systemic conflicts (e.g., in NN, NN only uses a quarter of the banks for the majority of read accesses), or the traffic is almost perfectly uniform, the benchmark will have a very different number of conflicts than what random accesses would result in. When calculating c_M for random accesses without a predetermined P, it is an optimization problem. The more requests there are, the more conflicts there will be, the longer the latency will be, and fewer requests within the system can be serviced. With too few accesses though, the memory system will be underutilized. The goal is to find the optimal number of requests and resulting number of conflicts for the system.

To find the probability of conflicts, we can view the problem as distributing all outstanding memory requests, o_M, randomly to all banks, b_M, and trying to answer the question what is the probability a request is assigned to a bank which already has at least 1 request? We can start with the expected number of banks to receive zero requests (Equ. 6.22). Every other bank is expected to receive at least one request (Equ. 6.23). Every request over the number of banks that receive at least 1 request, must encounter a bank conflict (Equ. 6.24). The number of requests that encounter a bank conflict is divided by the total number of requests to obtain the probability of a request encountering a bank conflict, c_M (Equ. 6.25).

$$\begin{aligned} \text{P[0 requests received at a bank]} &= (\frac{b_M - 1}{b_M})^{o_M} \\ \text{E[number of banks receiving 0 requests]} &= b_M(\frac{b_M - 1}{b_M})^{o_M} \end{aligned} \tag{6.22}$$

$$\text{E[number of banks receiving 1+ requests]} = b_M - b_M(\frac{b_M - 1}{b_M})^{o_M} \tag{6.23}$$

$$\text{E[number of conflicts]} = o_M - (b_M - b_M(\frac{b_M - 1}{b_M})^{o_M}) \tag{6.24}$$

$$c_M = \text{P[conflicts]} = \frac{o_M - (b_M - b_M(\frac{b_M - 1}{b_M})^{o_M})}{o_M} \tag{6.25}$$

The number of outstanding memory requests in the memory system, o_M is based on the average number of memory requests the ReRAM system is completing per cycle, P_R, and the average latency of the memory requests with all the contention, L_{RC}. This is because on average, each of the P_R requests will have been in the system for L_{RC} cycles, as will the next set of P_R requests on the next cycle. If this is to be sustained, there should be $L_{RC}P_R$ outstanding requests in the system.

$$o_M = L_{RC}P_R \tag{6.26}$$

The latency with contention is also dependent on the number of requests in the memory system. As the number increases, the contention and latency increases, based on the number of conflicts. To simplify the problem, we assume uniform traffic to each of the banks, and model the memory system as a M/D/1 queue. This means there is Poisson arrival times (which is expected for random traffic), a deterministic service time, and a single server. We selected the deterministic service rate because

we have exact service times with a known probability between the two, which is better approximated by a deterministic service rate than an exponential one.

The arrival rate, λ, for our system is the number of requests per cycle, P_R, divided between all the banks, b_M (Equ. 6.27). The service rate, μ, is 1 divided by the service latency, L_M (Equ. 6.28). The utilization, ρ, is the arrival rate over the service rate (Equ. 6.29). The average waiting time, w, in a M/D/1 queue is given in Equation 6.30, and is also our latency with contention, L_{RC}.

$$\lambda = \frac{P_R}{b_M} \tag{6.27}$$

$$\mu = \frac{1}{L_M} \tag{6.28}$$

$$\rho = \frac{\lambda}{\mu} = \frac{L_M P_R}{b_M} \tag{6.29}$$

$$w = L_{RC} = \frac{1}{\mu} + \frac{\rho}{2\mu(1-\rho)} = L_M + \frac{L_M P_R}{\frac{2b_M}{L_M} - 2P_R} \tag{6.30}$$

With the values of L_{RC}, o_R, and c_M established, they can be plugged back into Equation 6.21 to solve for the balance point. We were unable to find a closed form solution for this problem, so allowed a numerical solver to find the answer. Across the range of inputs, we find typical (i.e., 256 to 128k banks, 1 cycle to 2000 cycle latencies), the value of c_M was 0.39. If substantially different values are of interest, this calculation should be re-evaluated.

Finally, with the L2 cache sitting in front of the main memory, the number of memory accesses the main memory can support from the system's standpoint is

based on how many misses occur at the L2 cache, m_{L2}. If the miss rate is high, the main memory needs to be more parallel to support a high system parallelism; however, if the L2 miss rate is low, the memory system can still support a high system parallelism with little parallelism itself. Equation 6.31 describes this relationship and is the main memory parallelism, P_M, we expect in our system.

$$P_M = \frac{(1 - c_M)b_M}{\omega L_r + (1 - \omega)L_w} \times \frac{1}{m_{L2}} \tag{6.31}$$

While we have found the memory latency with contention, L_{RC}, this is particular to the optimal ReRAM memory system. We would like to find the latency with contention for reply packets based on how many replies the entire system will be sending. We again start from a M/D/1 queue. The arrival rate, λ, for our system is the number of requests per cycle, P_S reduced by the L2 miss rate, m_{L2}, divided between all the banks, b_M (Equ. 6.32). The service rate, μ, is 1 divided by the service latency, L_M (Equ. 6.33). The utilization, ρ, is the arrival rate over the service rate (Equ. 6.34). The average waiting time, w, in a M/D/1 queue is given in Equation 6.35. We replace the service time term with the read latency, L_r because we only care about the latency of read replies. This results in Equation 6.36 used to calculate our latency with contention based on the system request rate, L_{MC}.

Table 6.4: Main Memory Analytical Model Parameters

Variable	**Meaning**	**Unit**	**In/Out**
P_S	System Parallelism	Requests / cycle	Output
P_R	ReRAM Parallelism	Requests / cycle	Output
b_M	Main Memory Banks	-	Input
L_M	Base Main Memory Latency	cycles	Output
L_r	Read Latency	cycles	Input
L_w	Write Latency	cycles	Input
ω	Read percentage	%	Input
c_M	Main Memory Bank Conflicts	%	Output/Input
o_M	Outstanding Memory Requests	%	Output
L_{RC}	ReRAM Latency with contention	%	Output
P_M	Main Memory Supported Parallelism	Requests / cycle	Output
m_{L2}	L2 miss rate	%	Input
L_{MC}	Reply Memory Latency with contention	cycles	Output

$$\lambda = \frac{m_{L2} P_S}{b_M} \tag{6.32}$$

$$\mu = \frac{1}{L_M} \tag{6.33}$$

$$\rho = \frac{\lambda}{\mu} = \frac{L_M m_{L2} P_S}{b_M} \tag{6.34}$$

$$w = \frac{1}{\mu} + \frac{\rho}{2\mu(1-\rho)} \tag{6.35}$$

$$L_{MC} = L_r + \frac{L_M m_{L2} P_S}{\frac{2b_M}{L_M} - 2m_{L2} P_S} \tag{6.36}$$

Table 6.4 summarizes all the parameters and outputs of the main memory's model.

6.2.4 LLC Parallelism

The last-level cache has very similar characteristics as the main memory. Equation 6.37 gives the request parallelism of the cache, P_C, based on the total number of cache banks, b_C, the average cache latency, L_C, and the rate of cache bank conflicts, C_C.

$$P_C = \frac{(1 - c_C)b_C}{L_C} \tag{6.37}$$

The conflict calculation follows the same reasoning as it did for the main memory, as laid out in Equation 6.40. The number of outstanding requests is also in the same form, shown in Equation 6.39.

$$c_C = \text{P[conflicts]} = \frac{o_C - (b_C - b_C(\frac{b_C - 1}{b_C})^{o_C})}{o_C} \tag{6.38}$$

$$o_C = L_{LC} P_C \tag{6.39}$$

We again assume uniform traffic to each of the banks and use a queuing model to find the latency with contention. Instead of the M/D/1 queuing model, we use the M/M/1 queuing model. This means there is Poisson arrival times, an exponential service time, and a single server. We selected the exponential service time because the service time of cache accesses is more variable; this is because the cache requests have lower priority than the main memory requests for the unified controller. So even if the cache is ready to service another request, it may wait to do so when there

are many main memory requests finishing, adding to the effective service time.

The arrival rate, λ, for our system is the number of requests per cycle, P_C, divided between all the banks, b_C (Equ. 6.40). The service rate, μ, is 1 divided by the average service latency, L_C (Equ. 6.41). The utilization, ρ, is the arrival rate over the service rate (Equ. 6.42). The average waiting time, w, in a M/M/1 queue is given in Equation 6.43. This means that Equation 6.44 gives our latency with contention, L_{RC}.

$$\lambda = \frac{P_C}{b_C} \tag{6.40}$$

$$\mu = \frac{1}{L_C} \tag{6.41}$$

$$\rho = \frac{\lambda}{\mu} = \frac{L_C P_C}{b_C} \tag{6.42}$$

$$w = \frac{1}{\mu} + \frac{\rho}{\mu(1-\rho)} \tag{6.43}$$

$$L_{LC} = L_C + \frac{L_C P_C}{\frac{b_C}{L_C} - P_C} \tag{6.44}$$

We again substituted back the values of L_{LC}, o_C, and c_C into Equation 6.37 to solve for the balance point. We were unable to find a closed form solution for this problem, so allowed a numerical solver to find the answer. Across the range of inputs, we find typical (i.e., 64 to 2048 banks, 1 cycle to 20 cycles latencies), the value of c_M was 0.44. If substantially different values are of interest, this calculation should be re-evaluated.

For the latency with contention from the system point of view, L_{CC}, we can

Table 6.5: LLC Analytical Model Parameters

Variable	Meaning	Unit	In/Out
P_S	System Parallelism	Requests / cycle	Output
P_C	Cache Parallelism	Requests / cycle	Output
b_C	Cache Banks	-	Input
L_C	Average Cache Latency	cycles	Input
c_C	Cache Bank Conflicts	%	Output/Input
o_C	Outstanding Cache Requests	%	Output
L_{LC}	Cache Latency with contention	%	Output
L_{MC}	Reply Cache Latency with contention	cycles	Output

substitute the system requests, P_S, for the cache requests, P_C, in Equation 6.44 to obtain Equation 6.45.

$$L_{CC} = L_C + \frac{L_C P_S}{\frac{b_C}{L_C} - P_S} \tag{6.45}$$

Table 6.5 summarizes all the parameters and outputs of the last-level cache's model.

6.2.5 Request Return Rate

In Section 6.2.1, we introduced the idea that the system parallelism is limited by the rate at which requests are returned to the CPU. This is based on the number of outstanding requests tolerated by the system and the average latency of a blocking read packet, as shown in Equation 6.47.

Equation 6.46 was given for the average latency of the system. In subsequent sections, we've shown each of the subsystems' latency with contention is dependent

on the overall number of requests handled each cycle, P_S. This means P_S and L_A are dependent on each other and a balance point at steady state between average latency and system parallelism needs to be determined.

To find this balance point, we can replace the instances of P_S in L_A with the value given in Equation 6.47 and solve for L_A to find values for latency and then number of requests that satisfy the system of equations. Since we were unable to find a closed form solution, we used a numerical solver to find this solution.

$$L_A = 2L_{NC} + L_{CC} + L_{MC}m_{L2} \tag{6.46}$$

$$P_S = \frac{o_R}{L_A} \tag{6.47}$$

We found multiple potential balance points for each configuration; however, many are not reasonable. The network channel utilization imposes one known upper bound on P_S, presented in Equation 6.18. We also know that the maximum value of P_S is limited by the subsystems' parallelisms, as shown in Equation 6.48. As the system parallelism approaches the maximum parallelism of the main memory or cache, the latency of the limiting subsystem with contention approaches infinity. If the parallelism goes beyond this point, the equations are no longer valid, producing negative values. We can use this upper bound on P_S to bound L_A. This gives us a lower bound, which also captures the network bounds on channel utilization, as shown in Equation 6.49. Using this bound very often reduces the potential points to a single one.

$$0 < P_S \leq \min(P_N, P_C, P_M) \tag{6.48}$$

$$0 < \frac{o_R}{\min(P_N, P_C, P_M, R_p, R_m)} \leq L_A \tag{6.49}$$

This solution for the overall system latency, and system parallelism when combined with the number of outstanding requests tolerated by the CPU, is most often the limiting factor for the system. Since the latency of the memory, network, and last level cache tend toward infinite, the subsystems themselves will not be the limiting factor unless the program can send virtually infinite outstanding requests. The other limit on system parallelism can occur when the program is very compute intensive rather than memory intensive. We will see these limitations play out in Section 6.4.2.

6.3 Model Agreement

We tested the analytic model against the simulations we ran in Chapter 5. Our simulator captures all the needed workload dependent inputs for each benchmark. We use these characteristics and the configuration settings to populate the inputs of our model. Then the parallelism of the simulation is compared to expected parallelism according to the analytic model.

The baseline configuration and many of the parameter sweeps used to verify our model are presented in Table 6.6. The configurations include, at 256 tiles, sweeping ReRAM read latencies from 100 ns to 1000 ns with the write latencies

Threads	1024
Clock rate	2 GHz
L2 Cache Size	256 KB, 2 banks
L2 Cache Latency	1.8 ns average (pipelined)
Stride Prefetcher	32-entry stride table
Unified Controllers	256 (1 per core), 64 MSHRs
ReRAM banks	32,768 (128 per tile)
ReRAM read/write latency	200/400 ns
On-chip Network	16 x 16, 2D-Mesh
Network Channels	4 x 64 bytes

Table 6.6: Baseline simulation parameters used in Chapter 5.

double the read for 128, 256, 512, and 1024 threads, sweeping the write latency from 200 ns to 2000 ns with a fixed read latency for 256, 512, and 1024 threads and at 50 ns, 100 ns and 200 ns read latencies, sweeping the number of ReRAM banks from 32 banks/tile to 512 banks/tile for 256, 512, and 1024 threads, sweeping the width of the network from 8 bytes to 72 bytes, sweeping the size of the L2 cache from 64 KB/slice to 1024 KB/slice, sweeping the number of L2 cache banks from 1 bank/slice to 8 banks/slice. At 64 tiles, we include sweeping the number of ReRAM banks from 64/tile to 2048/tile at 64, 128, 256, 512, and 1024 threads, sweeping the size of the L2 cache from 64 KB/slice to 4096 KB/slice, and sweeping the number of L2 cache banks from 1 bank/slice to 8 banks/slice. This results in 130 data points for benchmarks with all results, and 98 data points for K-means which lacks 64-tile results.

The simulator captures the benchmark specific characteristics needed to find the read/write ratio, vector ratio, read and write instruction widths, average request width, average hops per packet, and, in the localized benchmarks, active prefetch streams and bank conflicts. The simulator also finds the MPKI and L2 miss rate

Table 6.7: Model Agreement

Benchmark	**% Error**	r^2
APSP	14.4%	0.906
BC	22.7%	0.865
PR	24.3%	0.720
SSSP	20.6%	0.837
DAXPY	12.7%	0.943
Kmeans	39.8%	0.432
NN	23.2%	0.825
PF	26.1%	0.821
All	22.3%	0.912

for each benchmark and configuration as the model does not include detailed cache effects to predict cache efficacy. Each simulation data point also includes the configuration details: threads, number of cache banks, number of ReRAM banks, ReRAM read and write latency, size of the network, and width of the network. Finally, we did not vary some characteristics like cache latency, prefetch depth, and cycle time, so these inputs remain fixed in our model.

Figure 6.1 shows the simulated parallelism vs the model's parallelism for all configurations as well as broken out per benchmark. The line in each graph represents where the simulation's parallelism equals the analytic model's expected parallelism. Table 6.7 reports the percent error and the r^2 value for the equality line for each benchmark and overall.

When looking at the agreement on a per benchmark basis, the model does best for DAXPY–a very regular streaming benchmark. DAXPY is very memory intensive. Its memory accesses stream through all the banks evenly, and the vast

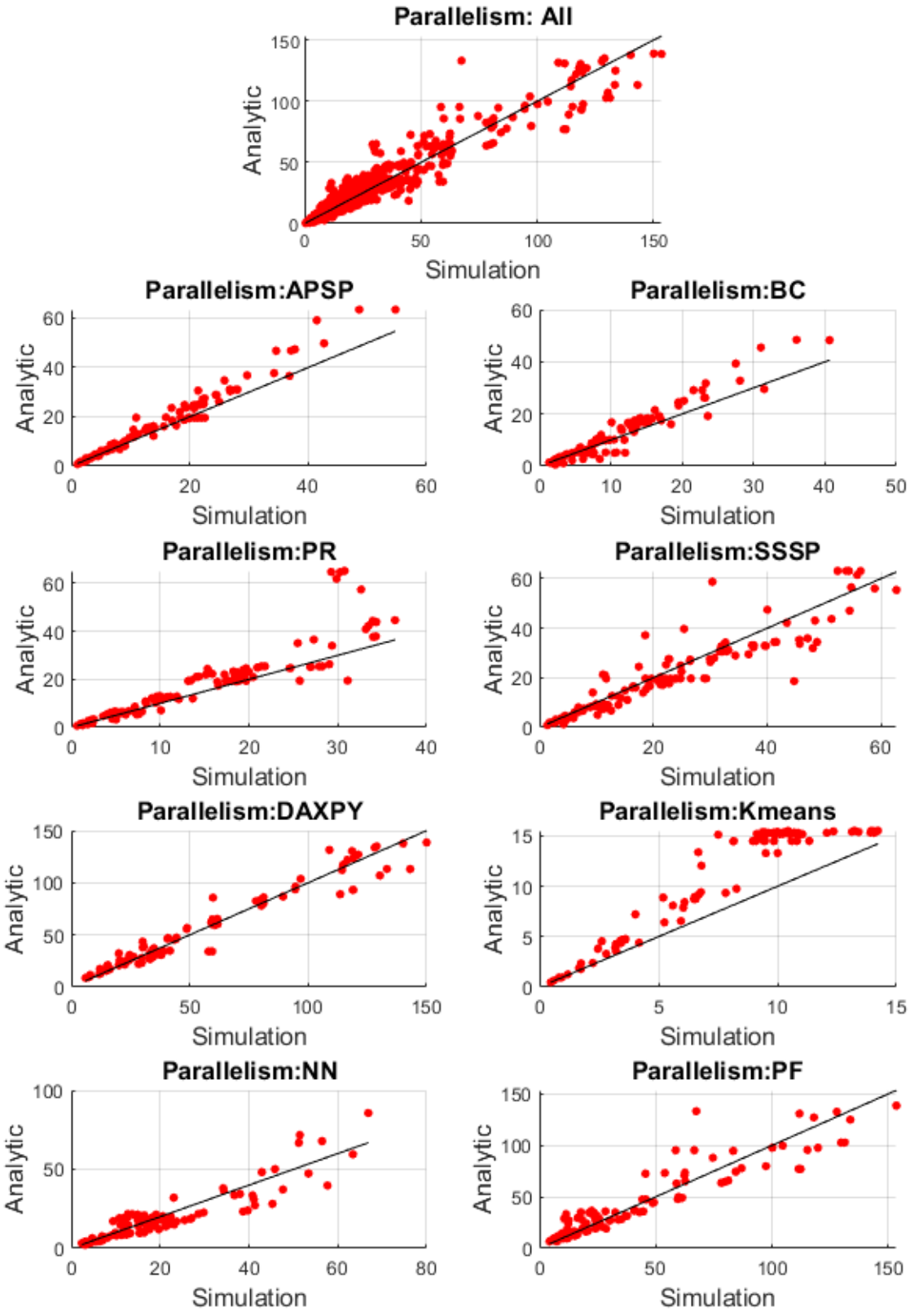


Figure 6.1: Analytical Model vs Simulation parallelism.

majority of its accesses are on the local tile. In the case of DAXPY, the r^2 value for the equal line is 0.942, and the average percent error for the analytic model compared to the simulation is 12.7%. Similarly, well performing is APSP, which has fairly evenly distributed requests throughout memory and time. For APSP, average percent error for the analytic model compared to the simulation is 14.4%

On the opposite spectrum, there is K-means, which has decent L1 cache performance, making the MPKI significantly lower than all other benchmarks. K-means average percent error for the analytic model compared to the simulation is 39.8%. The difference in performance may be due to not accounting for all sources of delay when a program is relatively compute intensive. Our model does not currently include things like context switching overhead which are included in the simulation model. PageRank (PR) includes a cluster of outliers where the ReRAM read latency was reduced to 50 ns and a sweep of ReRAM banks was performed. The performance of PR for these benchmarks was worse than PR with ReRAM read latencies of 100 ns. This is due to significantly more contention, suggesting systematic conflicts rather than only random conflicts as the analytic model assumes.

Overall, the analytic model shows good agreement with the simulated parallelism. The r^2 value for equal line is 0.912, and the average percent error for the analytic model compared to the simulation is 22.2%.

6.4 Architectural Balance

After developing the analytic model and confirming it gives reasonable outputs, we used it to map out the wider design space and identify the parameters that have the largest impact on the overall system parallelism. We also examine the architectural balance of the target system we evaluated in Chapter 5 on different types of workloads (streaming, graph, and compute intensive).

6.4.1 Design Space Overview

We did a large design space exploration of many model characteristics with a low, mid and high value. This gives us an idea of what the most impactful parameters are. Table 6.8 lists the parameters and what values they were given during the design space sweep.

Figure 6.2 shows the distribution of the configurations' parallelism. If any subsystem has very low parallelism, it will lower the whole system's parallelism, resulting in the high number of configurations with low parallelism. Whereas, every subsystem needs to support a high parallelism, resulting in a relatively low number of configurations with the highest parallelisms.

When examining the characteristics of the configurations with a parallelism of at least 1000 requests/cycle, we found that they all had the highest MPKI, outstanding prefetch requests, and threads from the compute side and the lowest ReRAM read latency and highest number of ReRAM banks on the memory side. All other parameters varied within this set. As the request return rate is generally the decid-

Table 6.8: Design Sweep Inputs

Variable	**Input**
Average Read Width	1, 4.5, 8
Average Write Width	1, 4.5, 8
Read Ratio	0, 0.5, 1
Vector Ratio	0, 0.5, 1
MPKI	0.1, 10, 1000
Streams	0, 5, 10
Cycle	0.5 (fixed)
Threads	32, 1040, 2048
Dimensions	2, 3, 4
Nodes	4, 18, 32
Prefetch Depth	32 (fixed)
Average Hops	1, 8.5, 16, and calculated for random traffic
Phits	1, 8.5, 16
ReRAM Banks	4096, 133120, 262144
ReRAM Read Latency	50, 1025, 2000
ReRAM Write Latency	1x, 3x, 5x ReRAM read latency
ReRAM Conflicts	0, 0.5, 1, and calculated for random traffic
Cache Banks	1, 8, 15
Cache Latency	1, 5.5, 10
Cache Conflicts	0, 0.5, 1, and calculated for random traffic
L2 Miss Rate	80 (fixed)

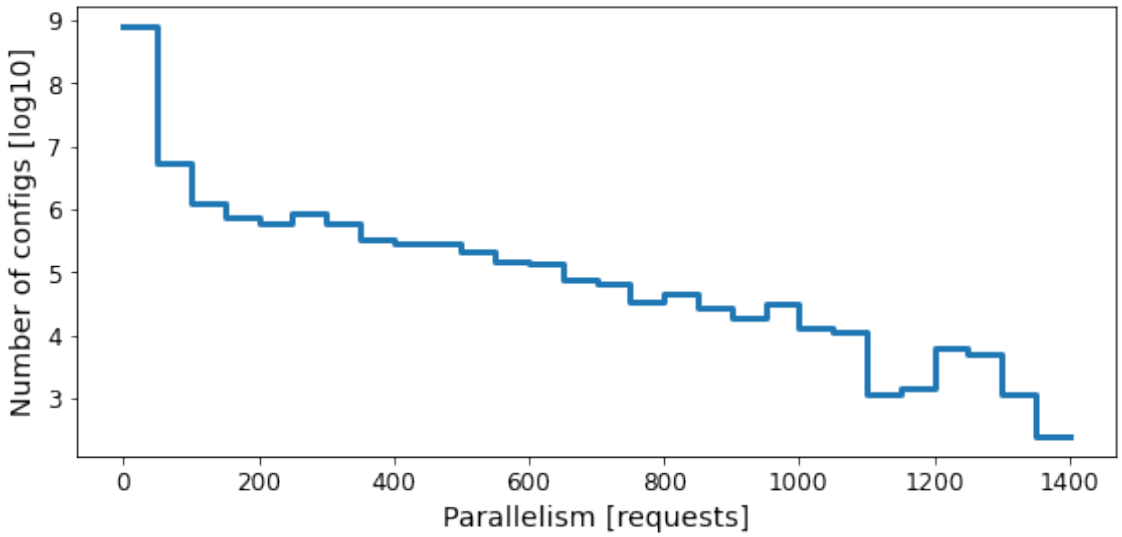


Figure 6.2: Design Sweep Points Parallelism

ing factor, this makes sense. The high amount of parallelism in the compute side ensures that o_R is as high as possible; the ReRAM latency is generally the largest contributor to L_A, so reducing the ReRAM read latency and any contention by increasing ReRAM banks will have the largest impact on L_A. Together, these have the most impact on creating the highest request return rates.

6.4.2 Case Studies

Beyond determining the most impactful parameters, we can also exercise the model to find specific configurations that are architecturally balanced. To demonstrate this, we explore finding architectural balance for a system starting from the system used in our simulations.

Using the baseline configuration in Table 6.6, we swept the parameters of the model to find a balanced configuration. Since all the subsystems' performance is dependent on workload characteristics, such as bank contention, read-write ratio,

and memory intensity, we selected 3 benchmarks to represent various workload types. Based on these workloads, we show where our system is over- or under-provisioned and which design decisions have the largest impact on system performance.

DAXPY is a very regular, memory-intensive, streaming benchmark. Most of its requests are vector requests to the local tile. The prefetcher works well and always has the maximum number of outstanding requests possible. It streams evenly through the ReRAM and cache banks, resulting in little to no conflicts. It is fairly write intensive, with about $\frac{1}{3}$ writes.

APSP is an irregular, memory intensive, graph benchmark. It reads from an edge array that works well with the prefetcher and does scatter-gather requests across main memory that do not. The prefetcher generally has a single outstanding request per thread. The number of network hops, cache conflicts, and ReRAM conflicts are average for random traffic. Over 90% of its memory accesses are reads.

Kmeans is not very memory intensive. It is a streaming benchmark, fetching primarily vectors; though since it isn't memory intensive, the prefetcher is turned off. It is fairly localized, but requests data from other tiles more often than DAXPY. Like APSP, it performs few writes.

Each of the graphs show the maximum bandwidth the system can sustain for the given configuration, "System," assuming 8B memory requests. The graphs also break down the maximum bandwidth for each of the subsystems, "Memory," "Cache," and "Network" and the bandwidth the program is requesting, "Program." The final line shows effective bandwidth of the system with round trip latency (the maximum bandwidth determined by the request return rate), "Requests." Each

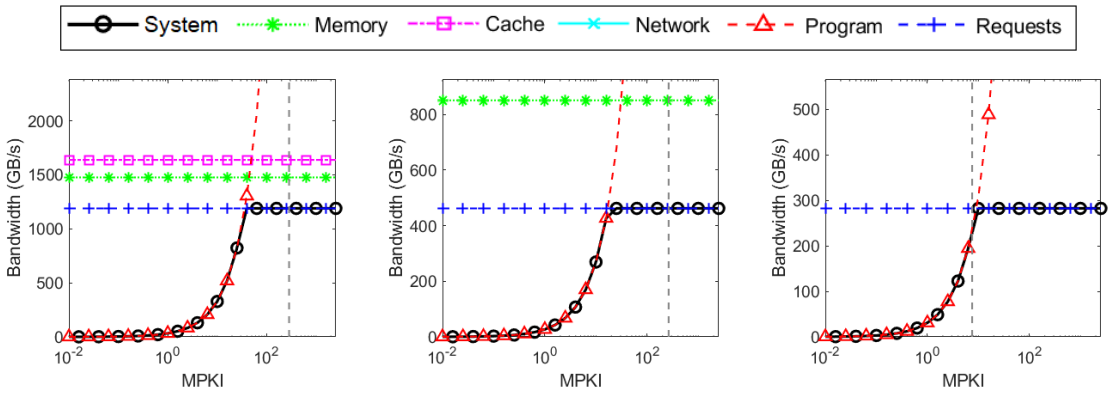
graph also includes a dashed grey line that denotes what value is used in the baseline configuration.

6.4.2.1 CPU and Program Characteristics

The parallelism from the compute part of the system is determined in part by the core architecture and the workload characteristics. These characteristics are things like the width of reads and writes, the ratio of reads to writes, the compute intensity, the average number of outstanding prefetching requests, and the number of threads. We will look at how the last 3 affect the parallelism of the system. This is because the compute intensity of the workload has a profound effect on the system bandwidth, and the prefetcher and number of threads are heavily influenced by the architecture.

While system architects cannot control the characteristics of the workloads, certain ones have a huge effect on the model and cannot be ignored. The memory vs compute intensiveness of the workload is one such parameter. A very memory intensive program will have a low memory bandwidth even if the system it is running on can support a higher one and will not benefit from many of the techniques we use to boost bandwidth. Instead, these benchmarks would need techniques focusing on increasing the core's efficiency to continue to increase performance.

We use misses per 1,000 instructions (MPKI) as our metric for how memory intensive a program is. Some programs may have very low MPKIs, such as 0.1 or 0.01, if they are compute intensive and their L1 cache has a high hit rate. The



(a) DAXPY-like (b) APSP-like (c) Kmeans-like

Figure 6.3: MPKI Sweep

theoretical maximum in our system is 8000, but that would require every instruction to be a scatter/gather instruction with 8 misses. Most of our benchmarks are in the 200–500 range; Kmeans is the exception with a MPKI of 7 on our system.

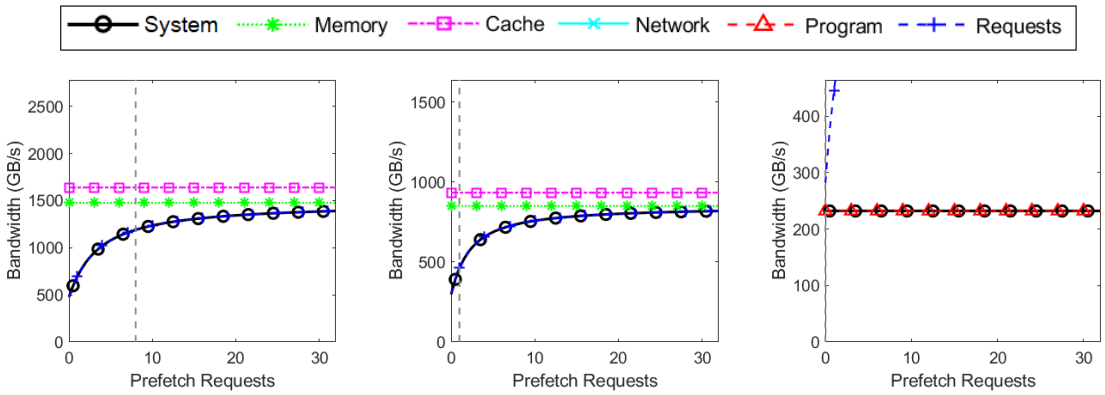
To see the effects of MPKI on system bandwidth, we swept the value from 0.01 to 2500 and plotted the results on a logarithmic plot in Figure 6.3. Sweeping the MPKI can be thought of as changing the number of non-memory instructions per memory instruction, while maintaining a similar memory access pattern otherwise. When the MPKI is low, the lack of memory requests is the limiting factor; then, the increase in requests rapidly increases the system bandwidth until it finally saturates the memory system.

The gap between where the program bandwidth crosses the requests bandwidth, and the gray dashed line indicating the baseline configuration's value, represents the mismatch between the compute parallelism and the memory system parallelism. For Kmeans-like programs, the two are almost perfectly balanced. For the more memory intensive benchmarks, the compute side is over-provisioned com-

pared to the memory system. In other words, the memory system is not meeting the program's required bandwidth and so cores spend time idle. For DAXPY-like programs, the compute system is about 8x over-provisioned, and, for APSP-like programs, it's about 12x.

In our model, MPKI is only used in Equ. 6.5; that equation also includes the average request size, v, and the number of tiles, N. We are using the number of tiles as a proxy for the number of 1 IPC cores in that equation. If we could vary the number of cores independently of the number of tiles and threads or number of instructions those cores issue per cycle, it would have a similar impact on the system bandwidth as varying MPKI. If trying to reduce just the compute capacity, care has to be taken not to reduce the total number of outstanding requests tolerated by the compute side (dependent on the number of threads) or the memory system's capacity (dependent on the number of tiles).

The number of outstanding prefetch requests is another characteristic that is highly influenced by the workload. Our system employs a stride prefetcher, so only regular striding accesses are able to be prefetched. How often the prefetch data is consumed limits the number of outstanding requests. If there are blocking memory requests that cannot be prefetched, they will prevent the use of already prefetched data until the blocking request is satisfied. Similarly, if the program has a high compute intensity, it will reduce how often the prefetched data is consumed. The number of hardware prefetch buffers places an upper limit on the number of outstanding requests from the prefetcher even in cases where the program is constantly using the prefetched data.



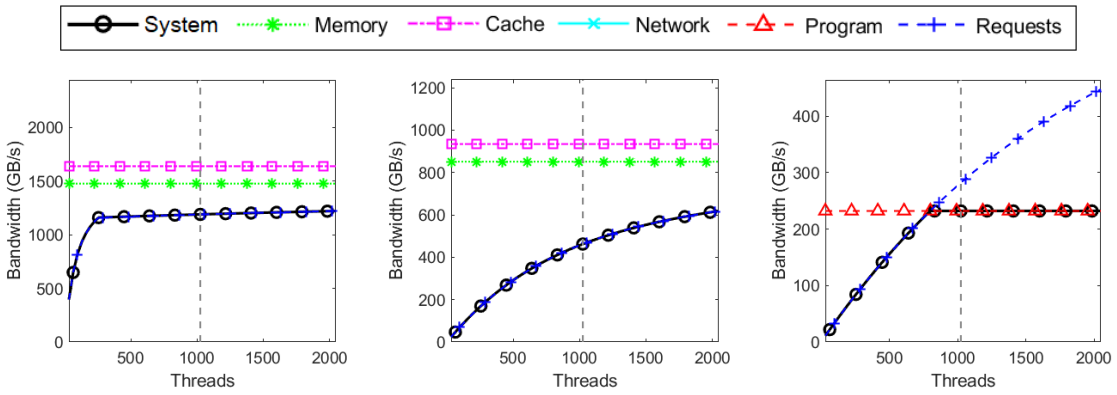
(a) DAXPY-like (b) APSP-like (c) Kmeans-like

Figure 6.4: Outstanding Prefetch Requests Sweep

Figure 6.4 shows the performance as we sweep the number of outstanding prefetch requests per thread in the system. A DAXPY-like program can benefit from more prefetching than the hardware in our architecture provided, though there are diminishing returns. The other types of workloads are limited not by the number of prefetch buffers, but the amount of prefetching the program supports. If indirect prefetching was implemented (i.e., a prefetcher that could prefetch memory requests in the form of $A[B[i]]$), the graph benchmarks could benefit, with the memory system eventually becoming the limiting factor with the increase in requests.

The final source of parallelism within the core we will look at is the number of hardware threads supported. Threads allow the program to continue execution of the program while waiting for memory requests to return. They also increase the total number of outstanding requests the system can tolerate from the compute side. We sweep the number of threads and present the system bandwidths in Figure 6.5.

As soon as there are enough threads to have a single thread per tile, a DAXPY-like program reaches a plateau. This is because the memory requests are driven



(a) DAXPY-like (b) APSP-like (c) Kmeans-like

Figure 6.5: Threads Sweep

primarily by the prefetcher which doesn't benefit from further threads but needs an active thread for each core to operate at maximum capacity (since the prefetch buffers are assigned per tile in our system). Kmeans increases until there are enough threads to hide the latency of the requests, at which point the program is compute bound rather than memory bound. Finally, a program like APSP continues to benefit from increasing the number of threads, though it is leveling off as each thread also increases the round-trip latency due to contention.

6.4.2.2 Network Characteristics

A CPU's network on chip is often overlooked in a shared memory computer. There are usually few enough messages that the network will rarely be a bottleneck. With our memory system, we can send so much data that the network starts to impact performance. For the simulation baseline, we configured the network such that it was not the limiting factor and left little room to try other configurations. With the analytic model, we look at a few changes to the network configuration and

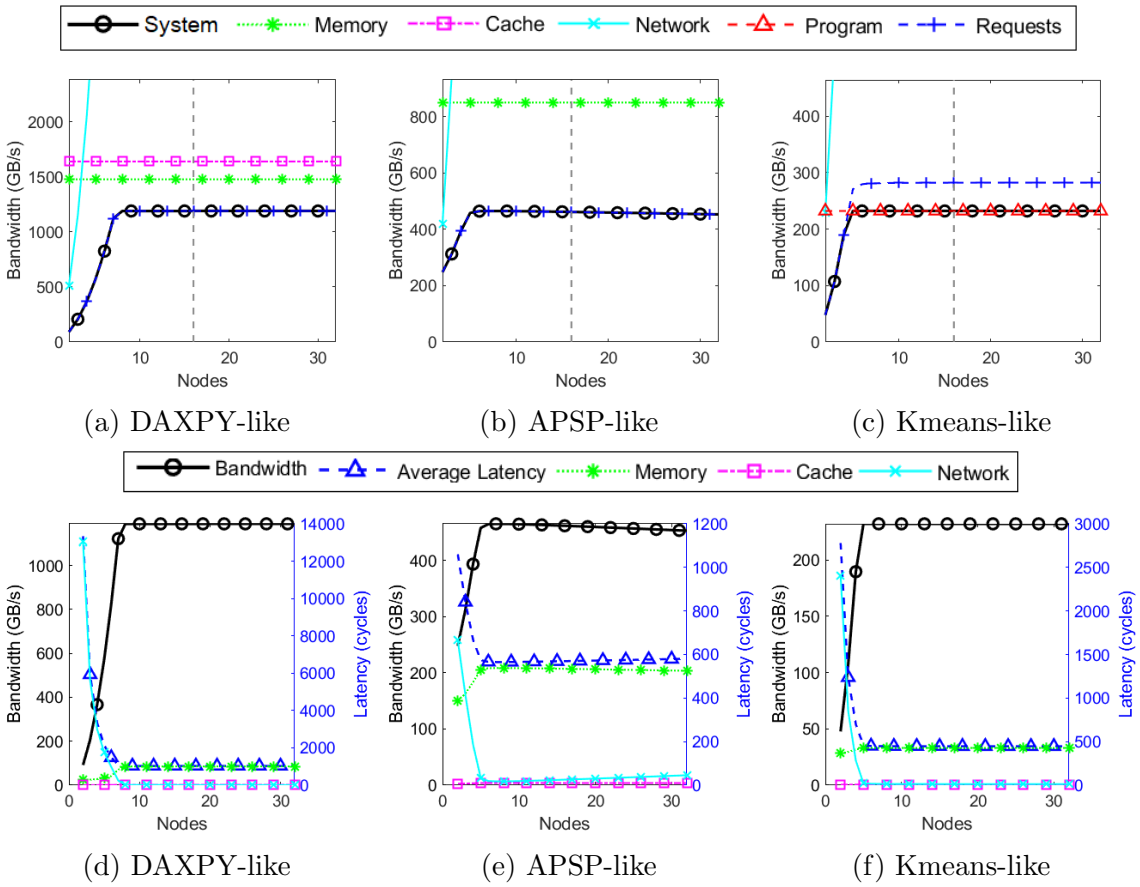


Figure 6.6: Network Nodes per Dimension Sweep

their impact on performance.

One possible network configuration of our tiled system would group multiple tiles into a single network node. To see the effects of this, Figure 6.6a–6.6c shows the bandwidth when the number of network nodes per dimension is varied, while leaving the number of tiles fixed. The baseline we use in our simulations is marked by the gray line at 16 with a 1-to-1 relationship with tiles. Figures 6.6d–6.6f incorporate a right-hand latency axis showing the average round-trip latency of the system, the latency within each of the subsystems. They also show the overall system bandwidth for the same configurations as Figure 6.6a–6.6c on the left axis.

Assigning a single network node to multiple tiles reduces the total number of

packets that can be injected into the network during a cycle compared to configurations with more nodes. It also reduces the overall capacity of the network, reducing its ability to handle requests in parallel. This is why all the benchmarks have poor performance when there are very few network nodes. However, cores generally do not send a packet during each cycle, so assigning network nodes to tiles in a 1-to-1 fashion often leads to under-utilization of the network.

As the number of tiles connected to a single node is reduced, eventually the network has the capacity to handle all requests. At that point, the network matters little to DAXPY and Kmeans, where the majority of the data is localized. For APSP, continuing to increase the number of nodes actually slightly decreases performance; this can be seen best in Figure 6.6e where the system bandwidth is no longer at its maximum and the network latency has increased by 32 nodes. APSP has random traffic; thus, as the number of nodes increases so does the average number of hops each packet will take, increasing network latency, and reducing overall bandwidth slightly.

For our system, this result shows that having groups of four tiles connected to a single network router (and subsequently reducing the number of nodes to 8 in each dimension), would not harm performance and may even increase it for the benchmarks which access memory locations spread across the chip.

An interesting thing to note in Figure 6.6d is how when the network latency decreases the memory latency increases. This is because the number of requests the system supports has increased, increasing queuing latency in the main memory. We can see the effect in reverse when we increase contention in a particular sys-

tem, decreasing the system bandwidth, and some subsystem's latencies will also be reduced. This emphasizes the interconnected nature of all the subsystems.

Making the network matter little to DAXPY and Kmeans took a fair amount of manual effort in localizing our streaming benchmarks. Localization will always be a benefit from the perspective of power as there is less data movement energy. It also means the capacity of the network can be reduced due to a reduction in occupancy by traveling packets. But does it actually improve performance within our system compared to the data being randomly distributed across the chip? We look at how changing the average number of hops per dimension affects the parallelism.

Figure 6.7 shows a sweep of the average number of hops in the system. Figures 6.7a–6.7c shows the bandwidth of the systems during the sweep. In addition to the gray dashed line representing the baseline value, we have added a purple dotted line to represent the expected number of average hops in our network for random traffic. Up to now, these graphs have been focused on the portion of the graph which shows the system bandwidth, but in Figures 6.7d–6.7f, we increase the range of the Y-axis to show the full network parallelism. Finally, Figures 6.7g–6.7i, show the latency of all the subsystems and the system bandwidth.

We can see that increasing the number of hops for Kmeans-like programs has little effect on the system bandwidth as the limiting factor is the program's required bandwidth rather than the system's ability to supply bandwidth. For APSP-like programs, the network's capacity is also adequate even when increasing the average number of hops, though performance does decrease somewhat as network latency increases.

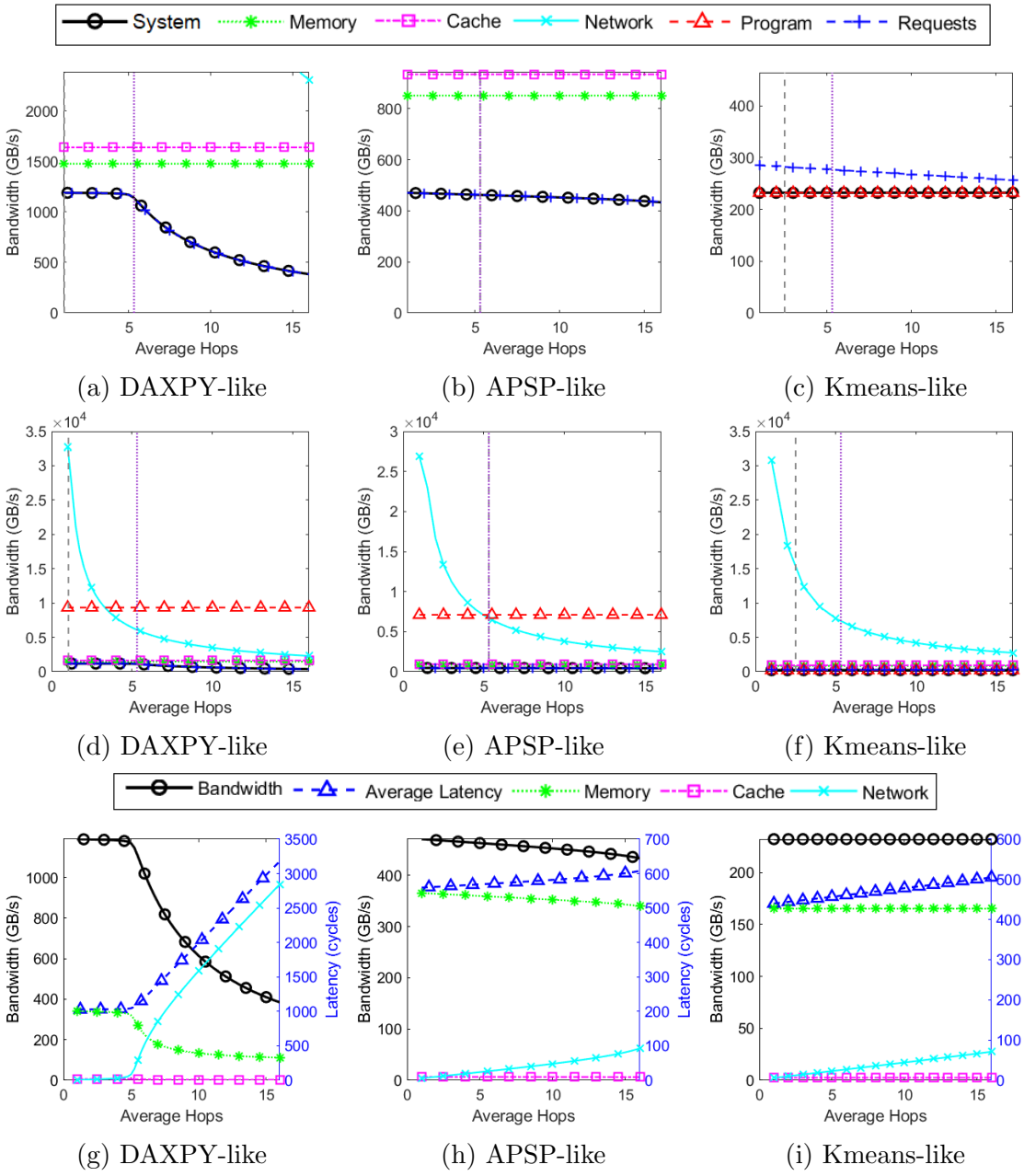
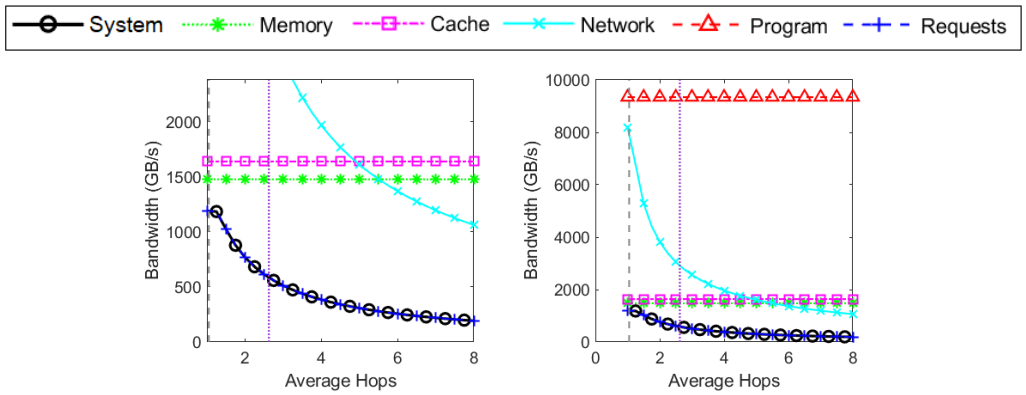


Figure 6.7: Average Hops Sweep



(a) Zoomed in bandwidth (b) Full Bandwidth

Figure 6.8: DAXPY Average Hops Sweep for 8×8 network

For DAXPY-like programs though, increasing the average number of hops can have a large detrimental effect on the performance of the program. DAXPY-like programs have the highest amount of memory parallelism due to their high degree of memory intensity and streaming nature enabling deep prefetching. As the network nears capacity, the network latency quickly rises as shown in Figure 6.7g. With our 16×16 network, the expected value for average hops with random traffic (the dotted vertical line) is about the limit before the network becomes a bottleneck in the system.

In Figure 6.8, we show the bandwidth of the system for a DAXPY-like workload as we sweep the average number of hops in an 8×8 network. Figure 6.8a shows the y-axis focused on the overall system bandwidth, while Figure 6.8b expands that axis to show the maximum bandwidth of all subsystems. This sweep shows that if we were to reduce the number of network nodes to 8 per dimension as suggested earlier, then not localizing DAXPY would put strain on the network.

Localization also helps with another potential problem that reduces perfor-

mance: cache and ReRAM bank conflicts. During the localization process, the data is carefully laid out to reduce contention; if data was distributed throughout the chip, streams from different threads may conflict with each other resulting in a rise in bank contention. We will look at how this contention affects system bandwidth in Sections 6.4.2.3 and 6.4.2.4.

Finally, another component of the network is its width. Within simulations, we've seen that with our 16×16 network, changing the network channel width had little effect. This is shown in the analytic model as well. But for the 8×8 network, there is some benefit to wider channels. We show the bandwidth of all the subsystems in Figure 6.9a–6.9c and the latency in Figure 6.9d–6.9f as we sweep the width of the network channels for an 8×8 mesh network.

The network bandwidth appears to be a stepwise function. This is because the model is based not on actual bandwidth of the network, but the number of memory requests the network can sustain per cycle, which is then converted into an effective bandwidth. Since there are different types of packets with different sizes, as each size of packet can be sent in a single cycle rather than multiple, the network parallelism increases, leading to the stepwise behavior. The smallest packets are read requests, which are 16-bytes. The largest are vector write packets at 80-bytes.

The network does need to be able to send all packets in a single cycle to no longer be the bottleneck, even for a benchmark with many vector writes, such as DAXPY. We can see that by a channel width of 40-bytes, the network is no longer a concern, even for DAXPY. Though this does present an interesting trade-off for network design within this system—are fewer nodes with wider channels or

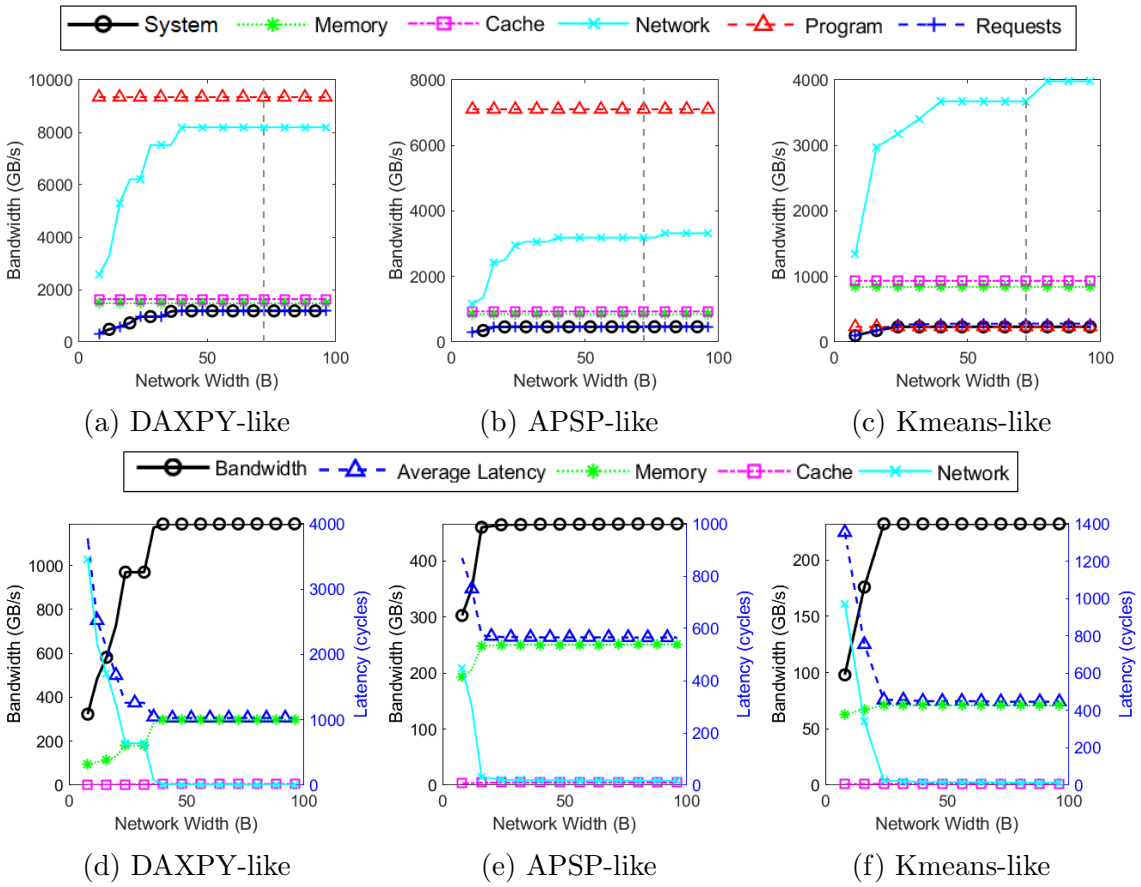


Figure 6.9: 8×8 Network Channel Width Sweep

more nodes with narrower channels more desirable? Both can achieve near optimal performance from a memory bandwidth perspective, so other considerations such as area and power will determine the best option.

6.4.2.3 Cache Characteristics

The effects of the cache and main memory parameters are more apparent than those of the network as our baseline configuration did not over-provision them to ensure they were not the limiting factor. Instead, the configuration was generally designed to stress these systems as much as possible. The characteristics of the

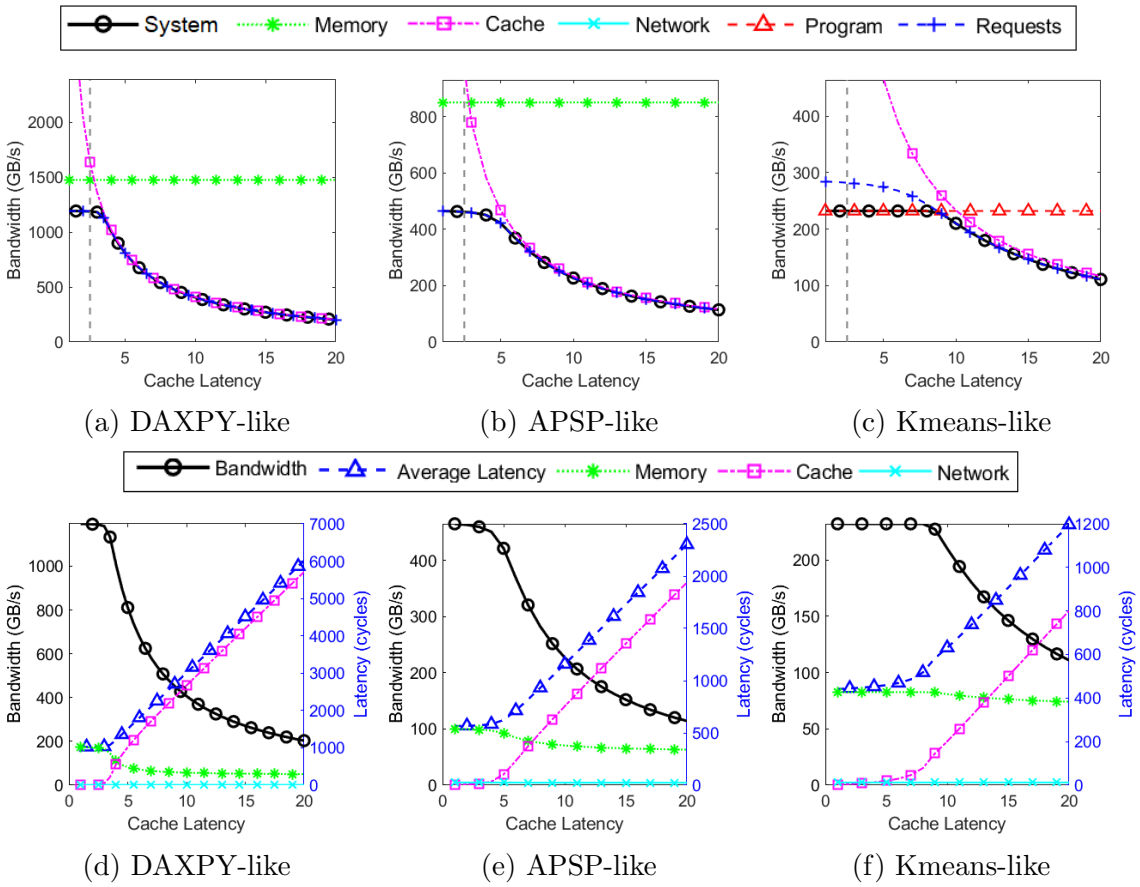


Figure 6.10: Average Cache Latency Sweep

cache with the greatest impact on the available bandwidth are the access latency, the number of cache banks, the amount of contention, and finally the miss rate. Characteristics like the cache capacity and associativity will impact the miss rate and access latency but are not directly used in the model.

The average latency of the cache has a large effect on the amount of available parallelism in the cache. Doubling the access latency results in halving the number of requests the cache can handle per cycle. In Figures 6.10a–6.10c, we see this play out as we increase the average access latency to 20 ns.

And as the access latency increases, the number of waiting requests and the time they spend in the queue increases rapidly as well. We can see the total average

latency of the cache in Figures 6.10d–6.10f, which climbs well beyond the nominal access latency of the cache.

If the LLC has this kind of latency, techniques to increase the parallelism of the cache will become essential to decrease the queuing latency. One option is pipelining, which would overlap requests and increase the rate at which requests are completed. Another option is increasing the number of cache banks, reducing the number of requests in each bank queue.

To show the effect of cache banking on our system, we swept the number of cache banks from 1 to 16. The bandwidth of the system is shown in Figures 6.11a–6.11c, with the latencies shown in Figures 6.11d–6.11f.

When we modeled cache banks in our simulator, we were modeling the MSHR file and increasing the number of cache banks reduced the number of MSHRs available per bank and could stall execution with unused cache banks if memory requests were unevenly distributed to each bank. The analytic model, however, assumes there are always an adequate number of MSHRs. Under this assumption, both DAXPY-like and APSP-like programs see an increase in performance from one cache bank to two cache banks, even with our baseline access latency (2.5 ns), at which point main memory becomes the limiting system. Kmeans is still limited by the program request rate with any level of cache banking in this system.

However, if the last level cache latency was higher, such as if lower power transistors were used to decrease static power, the number of cache banks has a more noticeable impact. Figure 6.12 shows the same cache bank sweep but with the cache access time at 10 ns rather than 2.5 ns. The incredibly high cache latency

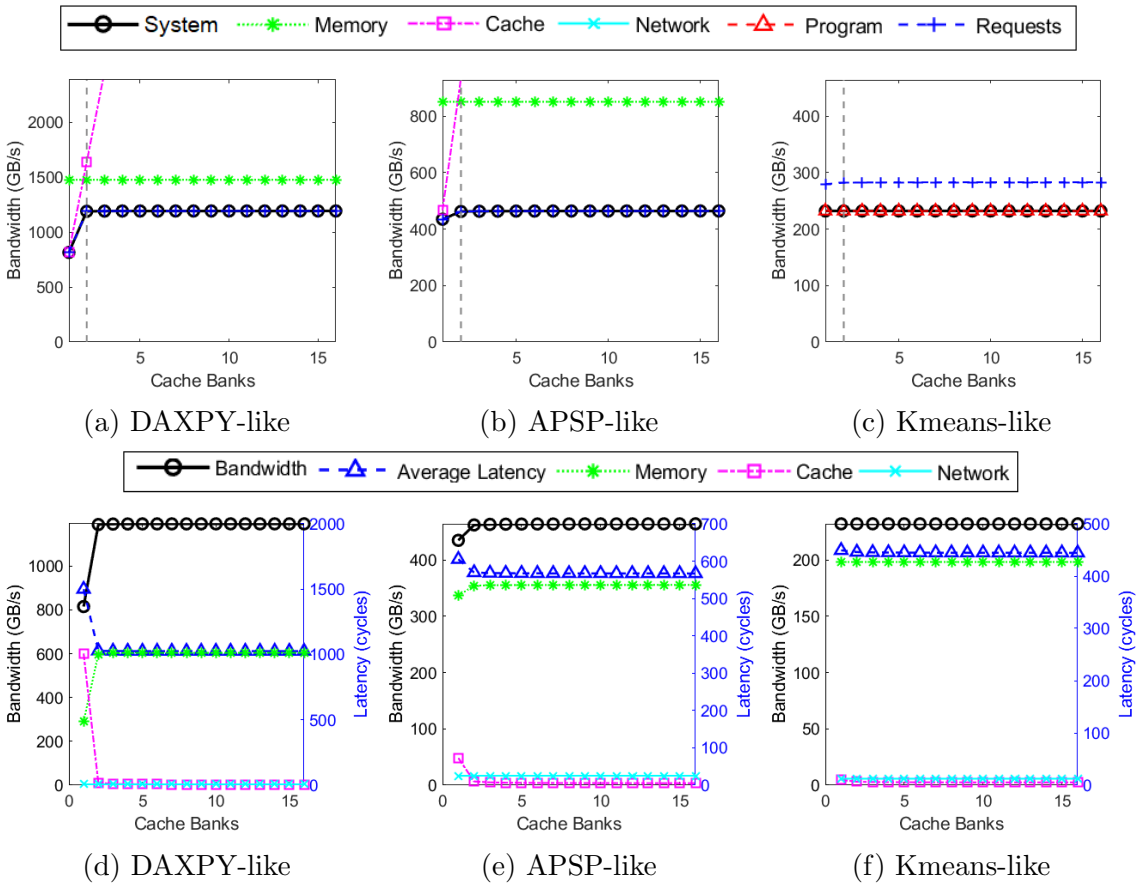


Figure 6.11: Cache Banks per Tile Sweep

in Figures 6.12d–6.12f shows the cache is very under-provisioned with only a single cache bank. With 10 ns access latencies, even Kmeans-like programs benefit from having multiple cache banks, and DAXPY-like programs require upwards of five for the cache to no longer be the limiting system.

The cache latency and amount of cache parallelism need to be balanced to create an optimal system. Low power transistors may save power within the arrays, but if more control circuitry is needed for multiple banks, then it may not be worth it.

The final parameter with a meaningful effect on the parallelism of the cache is the amount of bank contention. This occurs when requests are unevenly sent to

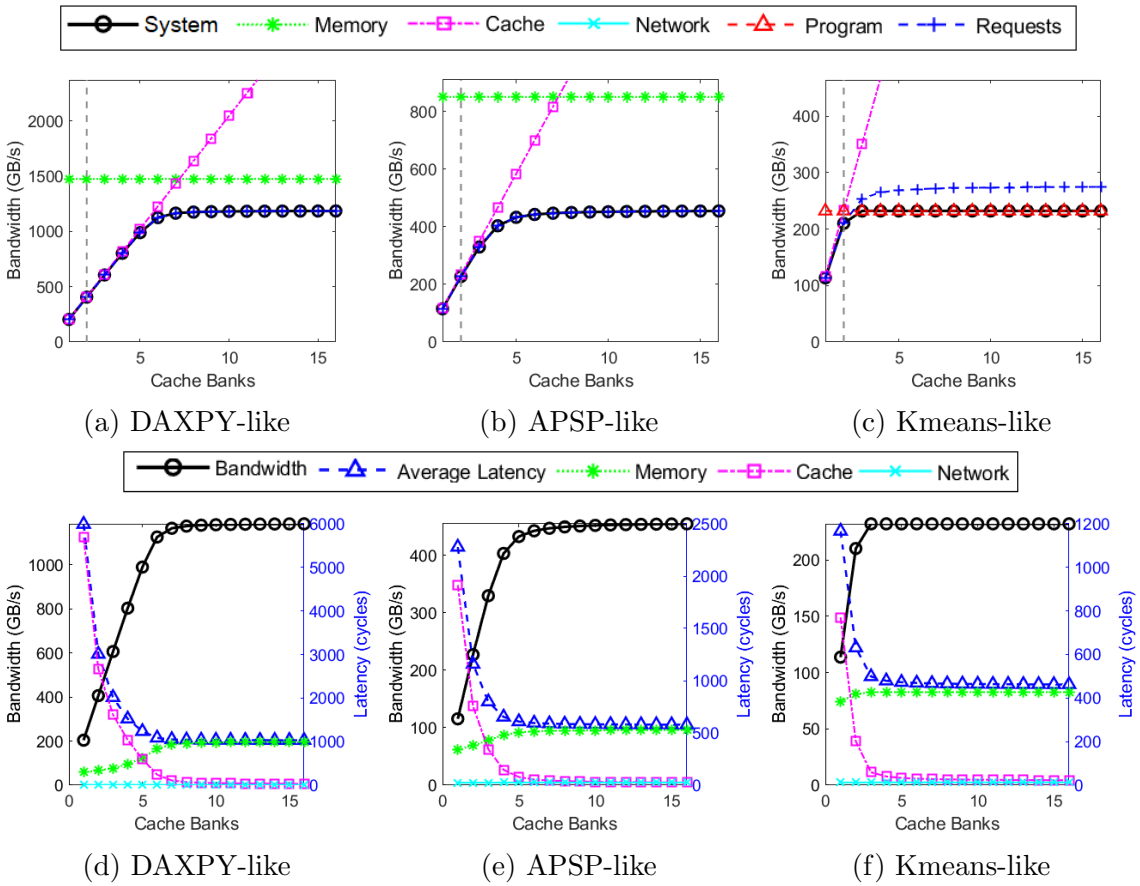


Figure 6.12: Cache Banks per Tile with 10 ns Access Latency Sweep

the cache banks, with some banks having many requests and others having none, effectively lowering the number of cache banks in operation. In the worst-case scenario, all traffic is sent to a single cache bank. This can occur (usually briefly) when all cores require the same piece of data, such as a base address stored in a global variable. Generally, phases of the program where every core requires the same data should occur rarely, and the bulk of the workload should be accessing data without systemic conflicts.

We look at how the amount of contention in the cache can affect the parallelism of our system. In Figure 6.13 we show the bandwidth and latency of the system as we increase the number of conflicts in the system from none to 100%.

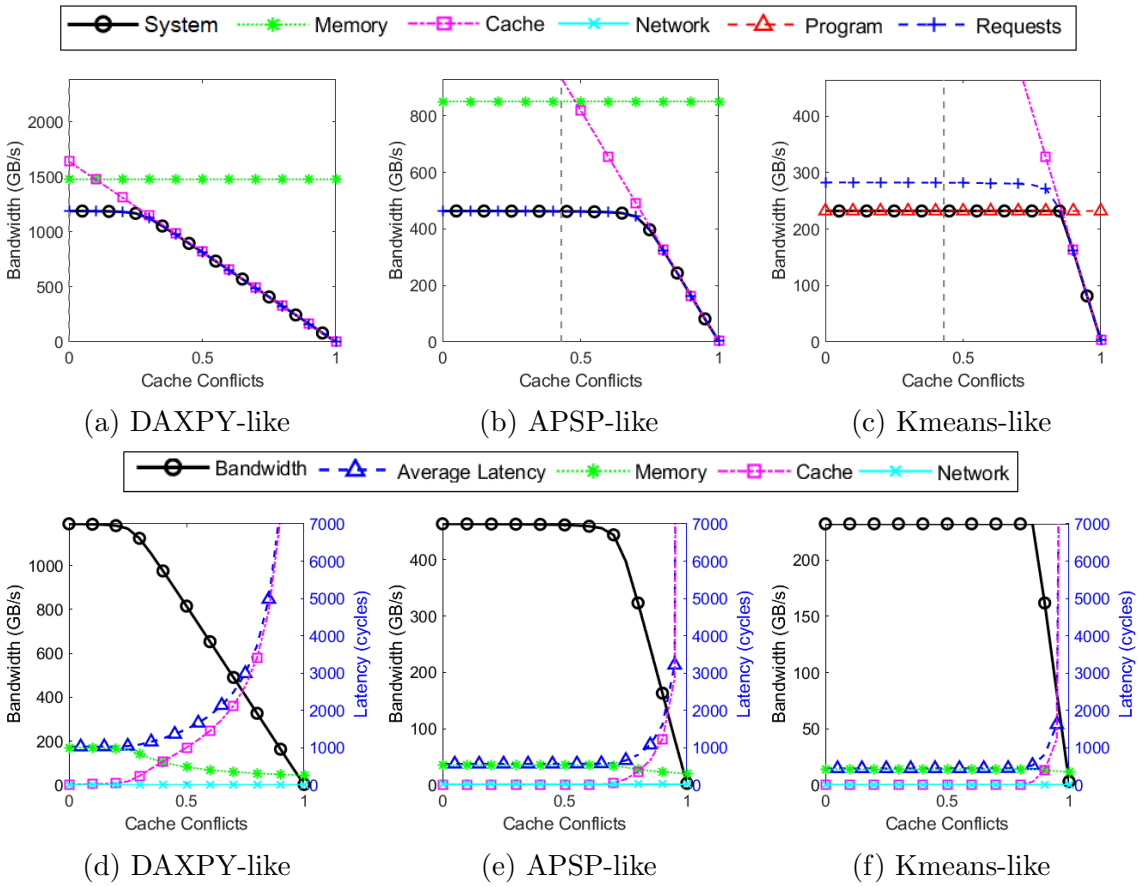


Figure 6.13: Cache Conflicts Sweep

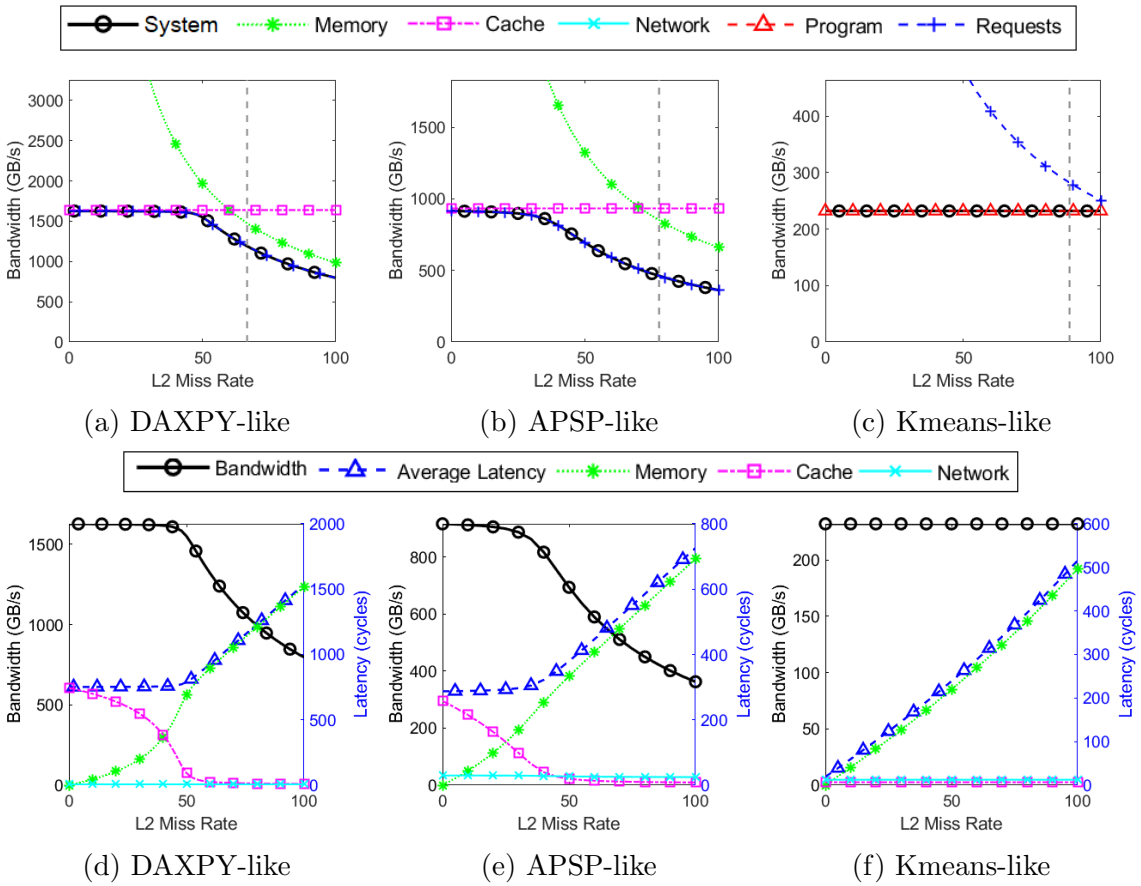
DAXPY-like programs are carefully localized and laid out such that bank conflicts are rare. The memory requests stream through each bank, ensuring they are evenly used throughout the program's execution. This sweep shows that the cache could handle a higher degree of contention without issue, though it does start to affect the performance of DAXPY-like programs around 30%. APSP-like and K-means-like programs are similarly not affected detrimentally by the amount of contention they see in typical runs, with some room for more contention. Both exhibit higher contention than DAXPY but have a lower sustained bandwidth due to other factors.

We can see on the far end of each of the graphs, if there are systemic issues

creating bank contention, the bandwidth of the system will fall sharply as the latency of the cache rises exponentially. For good performance in this system, it is vital that systemic contention does not occur.

While the LLC miss rate is closely associated with the cache design, it most heavily influences the main memory parallelism. If the LLC can filter out many of the requests to main memory, the memory can have less parallelism while still supporting a high system bandwidth. The miss rate is dependent on cache characteristics such as capacity, associativity, and replacement policy. However, it is also dependent on the workload, similar to MPKI, which characterizes misses over time in the L1 cache. For this sweep, we assume that the workload characteristics are very similar but its working set fits better or worse within the LLC. To examine the effects the LLC miss rate, we've swept it from 0% to 100% and report the results in Figure 6.14.

For the DAXPY- and APSP-like programs, we can see that increasing the LLC miss rate decreases the performance of the system. At low levels of misses, the cache is the limiting system. When there are no cache misses, the memory latency is effectively zero as no requests encounter it; the system parallelism is high, as well as the queuing latency at the cache, to take advantage of the lack of memory latency. As the miss rate increases, the memory latency rises. This causes a very small decreases in the system parallelism, but even small changes can drastically reduce the cache latency when it is near capacity, to keep a steady system latency. Eventually the ReRAM latency dominates, and the system parallelism decreases as the cache can no longer reduce its latency with small reductions in system parallelism. At this



(a) DAXPY-like (b) APSP-like (c) Kmeans-like

(d) DAXPY-like (e) APSP-like (f) Kmeans-like

Figure 6.14: L2 Miss Rate Sweep

point, the main memory is the limiting system.

6.4.2.4 Main Memory Characteristics

The main memory characteristics are very similar to that of the cache—just larger. The access latency is about two orders of magnitude larger, but so is the number of banks to balance it out. While the two are similar, in the majority of our benchmarks, for the base-line configuration, the main memory is the limiting system.

The ReRAM access latency is a major component of the main memory's avail-

able parallelism. The access latency for ReRAM reads and writes is asymmetric, so we look at varying both the read latency and the write latency independently. Since different workloads have different read ratios, the impact of each of the latencies will be workload dependent. For DAXPY, $\frac{1}{3}$ of the requests are writes, whereas for APSP and K-means, less than 10% are.

While part of the read latency is inherent to the cell and sensing circuitry, a major component is influenced by the ReRAM array size. Since read latency is dependent on the size of arrays, at what point the ReRAM access latency is no longer the limiting factor may influence which size of arrays should be selected by designers without needing to re-develop any of the more fundamental components like the cell.

To examine the effect of ReRAM read latency, we swept it from 50 ns to 2 μs while the write latency was held at 400 ns. The results of this sweep are presented in Figure 6.15.

For DAXPY- and APSP-like workloads, the cache is the limiting system at the lowest ReRAM read latency, but a cross over generally occurs before the baseline 200 ns read latency. The cache access latency drops as it is no longer the limiting system and the ReRAM latency rises even faster than its read latency, similar to what we saw in the cache access latency sweep (Fig. 6.10), as the amount of contention rises. A similar process occurs with Kmeans-like programs, though the cache does not begin as the limiting factor, but the program's required bandwidth.

Write latency is expected to be higher than read latency. Part of this is the fundamental time it takes to write the ReRAM cell, and part is due to schemes that

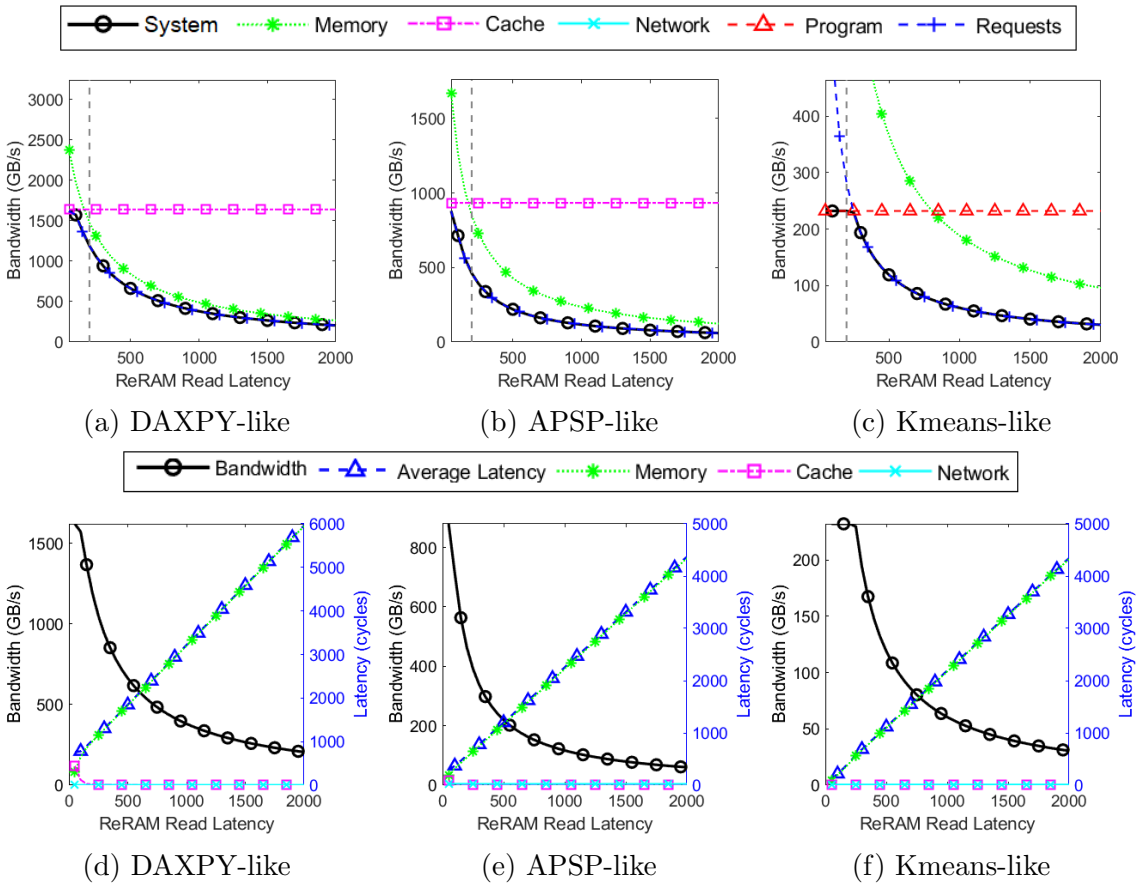


Figure 6.15: ReRAM Read Latency Sweep

write, verify the cell is written, and if not, write again. This will increase reliability of the ReRAM but may result in a high average write latency. The results for sweeping the ReRAM write latency from 50 ns to 5μs with a consistent 200 ns read latency are presented in Figure 6.16.

The results are fairly similar to the read latency sweep. In this case, only the cache in DAXPY is the limiting factor; read latencies of 200 ns even with a 50 ns write latency does not result in more parallelism to the memory system for a read intensive workload like APSP. Kmean-like workloads also have a longer time before the higher write latency affects their performance, with little impact even at 2 μs. As the write latency continues to climb though, it can be just as limiting as a high

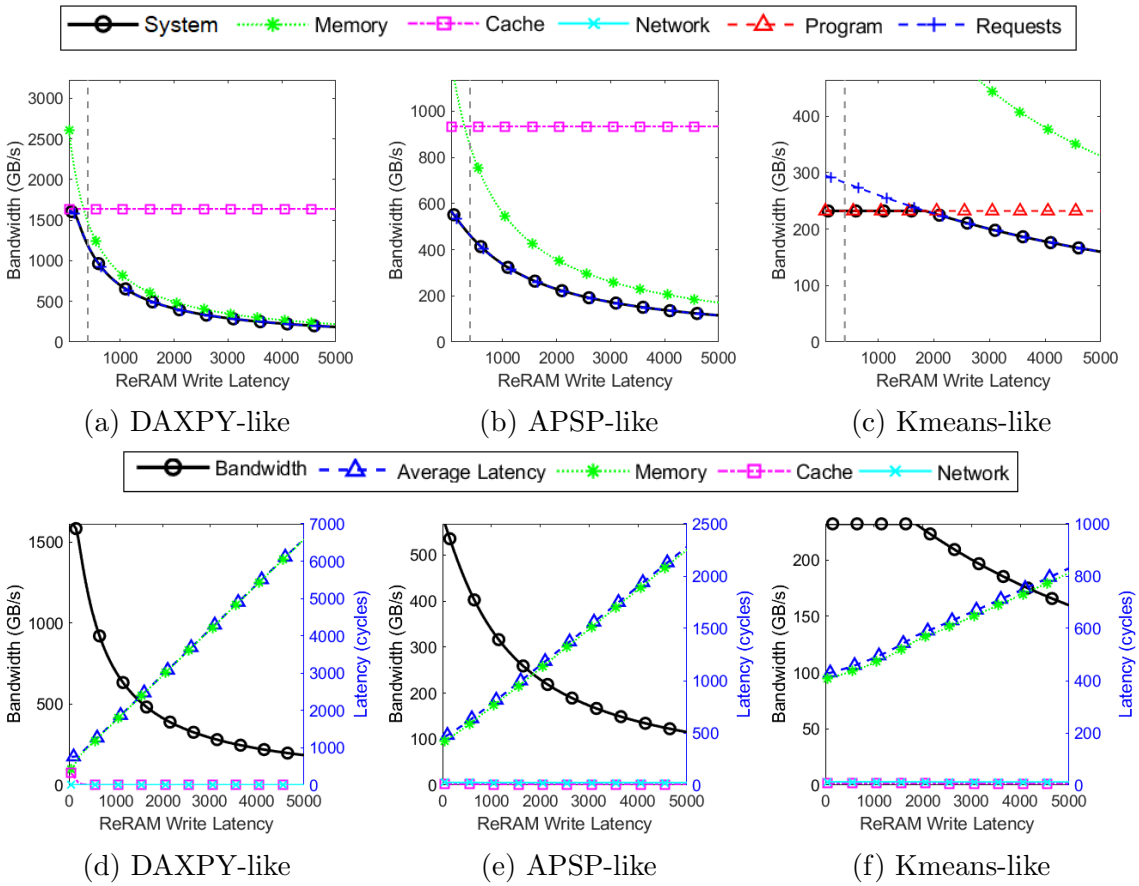


Figure 6.16: ReRAM Write Latency Sweep

read latency.

A high access latency can be compensated for with a larger number of banks to a point. This idea is how the ReRAM memory system is competitive with DRAM systems even with an order of magnitude higher access latency. In Figure 6.17, we present the results for sweeping the number of ReRAM banks per tile from 16 to 512, which represents 4,096 to 131,072 total ReRAM banks.

As expected, increasing the number of banks improves performance. For DAXPY-like workloads, at the low end of banks, the main memory is the limiting system but as the number of banks per tile increases, eventually the cache becomes a limitation. For APSP-like workloads, as the number of banks increases,

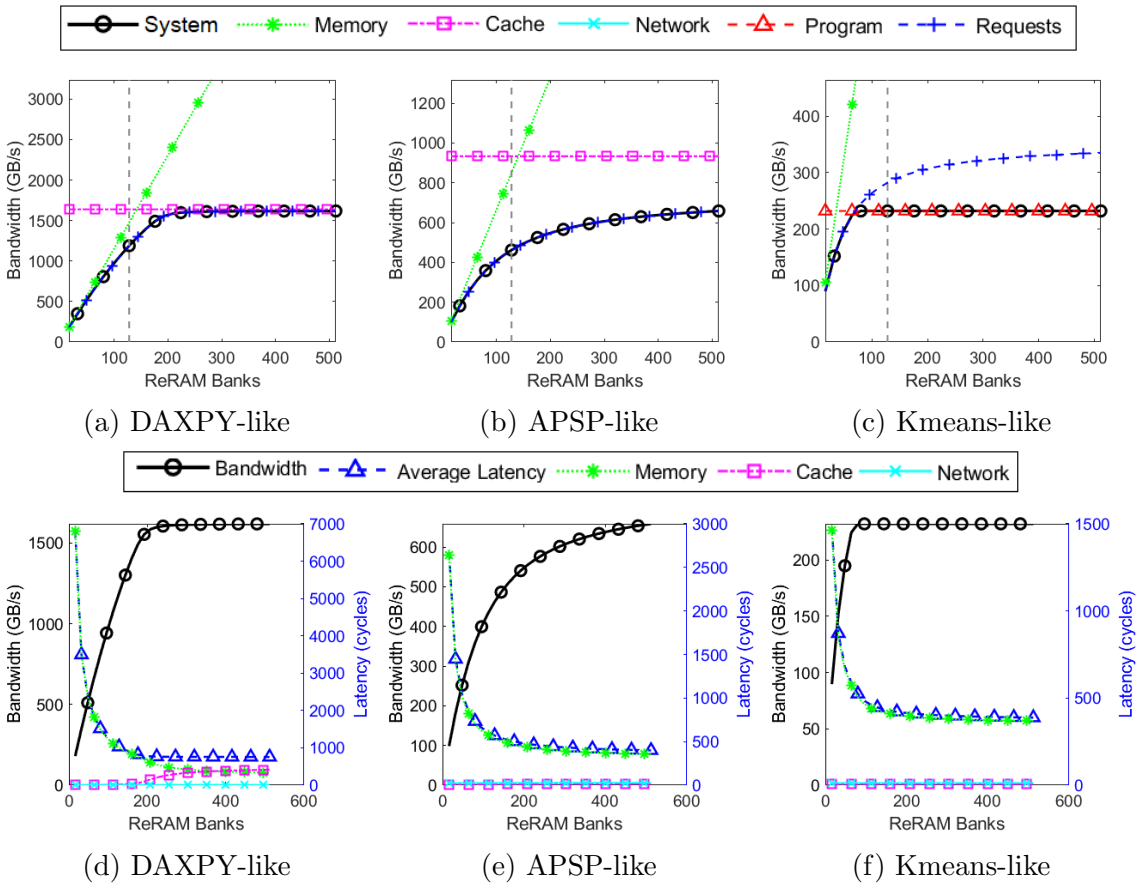


Figure 6.17: ReRAM Banks per Tile Sweep

the base system latency without contention is the limiting component. No subsystem is overwhelmed, but there is not enough parallelism in the compute side to hide all the memory latency. Increasing the number of threads, implementing indirect prefetching, or an out of order core would result in continued bandwidth increases as the number of ReRAM banks rise. Finally, K-means saturates around 64 banks per tile then benefits little from further increase in banks as it is compute intensive.

The final parameter is bank contention within the ReRAM memory system. Like the cache, bank contention occurs when requests are unevenly sent to the memory banks, resulting in certain banks having many requests and others having none. In Figure 6.18 we show the bandwidth and latency of the system are affected

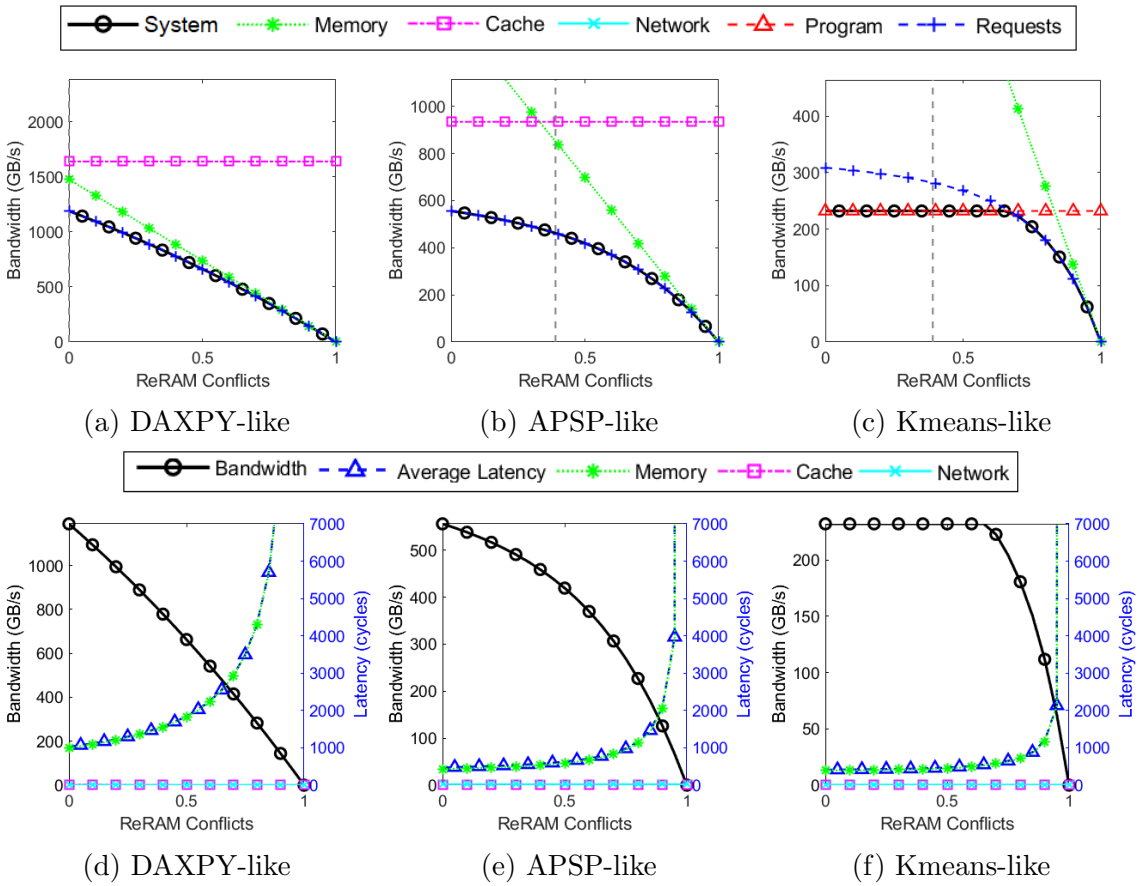


Figure 6.18: ReRAM Conflicts Sweep

as we sweep the amount of contention in the memory system from none to 100%.

At low amounts of contention, APSP-like and Kmeans-like workloads are not constrained by the ReRAM parallelism. DAXPY is still limited by the ReRAM parallelism even with few conflicts, though it is not on the steep part of the rising latency yet. This is partially why the localization which helps minimize bank conflicts improves DAXPY-like programs' performance.

At the far right of the graphs is the worst-case scenario where all traffic is sent to a single ReRAM bank. This is something we saw when first developing the simulator and had no shared last level cache. All the cores would nearly simultaneously request common data, like a base address and execution would grind to a halt as

the ReRAM bank conflicts rose to nearly 100%. This happened at every barrier as we flushed the L1s and re-requested common data like base addresses. While we cannot completely prevent this kind of access pattern within the cache, the shared LLC should mean that this should not occur within the main memory system.

6.5 Conclusion

In this chapter, we have developed an analytic model to allow us a wide design space exploration without requiring days of simulation for each design point. Developing this model showed the bandwidth of the system is driven either by the amount of bandwidth required by the system, or by the round-trip latency and number of outstanding requests the system can tolerate.

With the design space exploration, we found the key parameters that affect performance are the workload's MPKI which drives the amount of bandwidth required by the program, and the number of streams and threads to increase the number of outstanding memory requests, as well as ReRAM access latency and number of banks to service those requests quickly. All configurations with the highest bandwidths had the best values for these parameters while other inputs varied in that set.

Starting from the target architecture we developed and simulated in Chapter 5, we looked at what modifications would result in architectural balance for various workloads. For the memory intensive benchmarks, the compute side had 8x–12x more compute resources than the memory system could support. Reducing the

number or speed of the cores while not reducing the number of outstanding requests tolerated and memory parallelism could bring the system into better balance. We found the network was over-provisioned and could be reduced to an 8×8 network with 40 B channels rather than 16×16 network with 72 B channels. The cache generally has slightly more parallelism than the main memory; this higher parallelism could allow power gating to be used, increasing the cache latency while lowering power. The system without those changes, could support a higher bandwidth if the main memory had a lower latency or more banks. Though eventually workloads like APSP would likely need to increase the number of outstanding memory requests it tolerates through more threading or an indirect prefetcher to benefit from many more memory banks.

This model gives a good starting point for understanding the performance of the system as a whole. Future work could combine this performance model with cost models like area and power to allow a constrained design space exploration looking at trade-offs between the components.

Chapter 7: RTL Implementation

Up to this point, we have focused on the impact of monolithic ReRAM on very parallel manycore systems. Using simulation and analytic modeling to analyze the performance of the system, we've shown the performance improvement for extremely parallel, memory intensive benchmarks. Two important directions for this work are examining the monolithic ReRAM's impact on compute intensive benchmarks and creating a test chip to evaluate a monolithically integrated main memory system. To further both goals, we've developed an FPGA emulation platform of our system.

Within our simulator, we do not simulate workloads like the SPEC CPU 2017 benchmark suite [111] because we have a very simple core model that is not a good fit for such types of benchmarks. The FPGA implementation enables us to evaluate benchmarks that benefit from more complex core models such as out-of-order cores and cache coherency. To model the performance of our system, we develop an RTL model that alters requests to the FPGA's DRAM-based main memory to be at the speed of ReRAM requests. As we are only masking the latency of the FPGA's DRAM in the emulation platform, we cannot emulate benchmarks where ReRAM outperforms DRAM, but we can test the impact of higher latency on more latency bound and compute intensive applications.

Additionally, creating the cache controller and ReRAM RTL implementation gives us insight into the increased resources required when integrating the ReRAM controller into the cache controller. This is the main aspect of the cache which we could not analyze with Cacti.

Finally, we are looking to eventually fabricate a test chip for this system. While this is currently a long-term plan, this SoC implementation is an important first step toward making that a viable path.

7.1 Architecture

In order to create an emulation platform, we required an open-source RTL core which has compiler support for its instruction set architecture (ISA). We started by selecting the RISC-V ISA [112], a popular free and open RISC ISA. The RISC-V ISA was developed particularly as a streamlined, modular ISA for collaborative and open development. The RISC-V Foundation consists of over 300 member companies resulting in a very well supported and established standard. The ISA has compiler support, simulators, and many open-source implementations of systems using the ISA.

Of the open-source implementations, Berkeley's Rocket Chip is a well-supported system-on-chip (SoC) RISC-V implementation [113]. The Rocket Chip SoC is a spiritual successor of the Research Accelerator for Multiple Processors (RAMP) project [114] which aimed for architectural research to use FPGA-based emulation techniques. The Rocket Chip SoC aims to provide a useful framework for archi-

tectural research at the RTL level. Rocket Chip is implemented in a higher level hardware description language, CHISEL, which is compiled into Verilog. The framework around the SoC provides tools for RTL simulation, FPGA-accelerated simulation, VLSI flows for ASIC development, and workload generation. The system provides everything required for system-on-chip platform emulation, and hopefully fabrication.

The Rocket Chip SoC implements compute tiles containing a core and L1 caches, an interconnect to the L2, an L2, and a TileLink network to other peripherals. They have multiple RISC-V cores to choose from, including an out of order core, the Berkeley Out of Order Machine (BOOM), which is well suited to testing computationally intensive benchmarks. For our base system, we used the Dual Small BOOM Cores configuration that is one of the defaults provided by Berkeley. Table 7.1 provides the base configuration parameters. The BOOM core supports the RISC-V 64-bit ISA (RV64I) with the integer multiplication and division (M), atomic instructions (A), single and double precision floating-point (FD), and compressed instruction (C) extensions. Additionally, the BOOM team has plans to implement the vector extension [115]. The Rocket Chip platform is very configurable so all of these parameters can be tweaked and components like hardware prefetchers can easily be added.

For the DRAM only system, we used the included SiFive Inclusive L2 Cache [116], the standard L2 cache for the Rocket Chip system. When emulating the monolithic system, we modify this cache model to include a ReRAM controller which all memory requests pass through. This maintains the current DRAM memory interface,

Table 7.1: Dual Small BOOM Cores configuration

Cores		2 BOOM
	ISA	RV64IMAFDC
	Issue Width	3
	Target Clock Speed	1.6 GHz
	ROB Entries	32
	Physical Integer Registers	52
	Branch Predictor	TAGE
L1 I-Cache		
	Size	16 KB
	Associativity	4
	Block Size	64 B
	I-TLB Entries	32
L1 D-Cache		
	Size	16 KB
	Associativity	4
	Block Size	64 B
	MSHRs	2
	D-TLB Entries	8
L2 Cache		
	Size	512 KB
	Associativity	8
	Block Size	64 B
	MSHRs	8

but the added ReRAM controller time gates the requests as they return. This should be sufficient for latency bound computations, as, in general, ReRAM latency is much higher than that of unloaded DRAM. If this is eventually used for fabrication, the control of the ReRAM arrays will likely need to be updated to match the specifications required by the ReRAM arrays available for fabrication.

The ReRAM controller used in the emulation platform is a simple controller with no re-ordering. There are 8 independent ReRAM banks with 64-byte fetch

widths. We selected 8 banks to match the number of MSHRs in the L2 cache. Since the banks have 64-byte fetch widths, 8 banks are equivalent to 64 banks with 8-byte fetches as described in Chapter 3. If using 2k×1k ReRAM subarrays, this matches the number of ReRAM subarrays we can integrate over a 512 KB SRAM cache. Each bank is fed by a 4 deep request queue. If any queue fills, the L2 cache can no longer issue any requests until there is a vacancy. Since the system's target clock cycle is 1.6 GHz, the read latency is 320 clock cycles, equivalent to 200 ns, and the write latency is double that at 640 clock cycles, equivalent to 400 ns.

The system definition does not dictate the main memory configuration, instead using the standard Xilinx interface when generating the RTL for FPGA use. Generally, the main memory for an FPGA is predetermined by the specific board it is on.

7.2 FPGA Simulation

There are inaccuracies introduced by running on an FPGA platform rather than an ASIC; for example, FPGA clock rates are generally an order of magnitude less than 1.6 GHz. Because of this, there are tools to accurately simulate the target system on an FPGA. The Rocket Chip platform includes the tool FireSim which creates a simulation platform from the Rocket Chip SoC [117]. These FPGA accelerated simulations allow us to implement the design on hardware and run benchmarks at much nearer real speeds while generating performance metrics and debug information.

FireSim is closely integrated with Amazon Web Services (AWS). AWS is a cloud service where virtual computers can be rented on-demand. In addition to general purpose computers, they also offer large FPGA boards and the licenses for the software required to generate the FPGA images. FireSim provides a framework for automatically launching and flashing these FPGAs, copying the OS image, as well as starting workloads, and copying back results from the FPGA to the host computer.

In order to emulate the main memory system, FireSim uses a FASED (FPGA-Accelerated Simulation and Evaluation of DRAM) memory model [118]. It simulates DDR3 behavior independently of the FPGA's actual memory characteristics. The particular configuration for the DDR3 memory used in our simulation is 16 GB with 8 banks, 4 ranks, a read and write depth of 4 and first ready, first come, first served scheduling [119].

When generating our system, we included performance counters but did not include any of the debugging tools for the final system. The resulting simulated system ran at about 50 MHz (it varies slightly between runs), which is four orders of magnitude faster than our manycore simulator.

Included in FireSim is the support to run part of the SPEC CPU 2017 benchmark suite. Currently all of the integer benchmarks are supported. The tool, Speckle, helps to compile and run SPEC for the RISC-V ISA. Speckle uses GCC and the RISC-V tool chain to compile the target binaries and generates the run scripts and working directories for each of the benchmarks. An Arch Linux image is created using the tool FireMarshal. FireSim creates root file systems that are

Table 7.2: Instructions Executed (Trillions)

600.perlbench_s	ref	2.66
623.xalancbmk_s		1.30
631.deepsjeng_s		2.27
605.mcf_s	train	0.134
620.omnetpp_s		0.127
641.leela_s		0.367
648.exchange2		0.335
657.xz_s		0.123

applied to the Linux image with the files for each SPEC CPU 2017 benchmark. When FireSim launches the workload, it loads the correct file system and starts the benchmark.

7.3 Results

To evaluate the impact ReRAM has on latency-bound and compute intensive benchmarks, we ran the SPECspeed 2017 Integer benchmark suite. SPECspeed 2017 has parallelized some benchmarks using OpenMP; as we are using a dual core system, we ran benchmark each with two threads where supported. Due to financial constraints, we used the reference inputs for three benchmarks, and the training inputs for the remaining benchmarks. Unfortunately, 602.gcc_s and 625.x264_s failed to work on our system. The input and number of instructions executed for each benchmark is listed in Table 7.2.

We ran each benchmark on the DRAM system and on the emulated ReRAM system. At the end of the run, FireSim reported the number of execution cycles

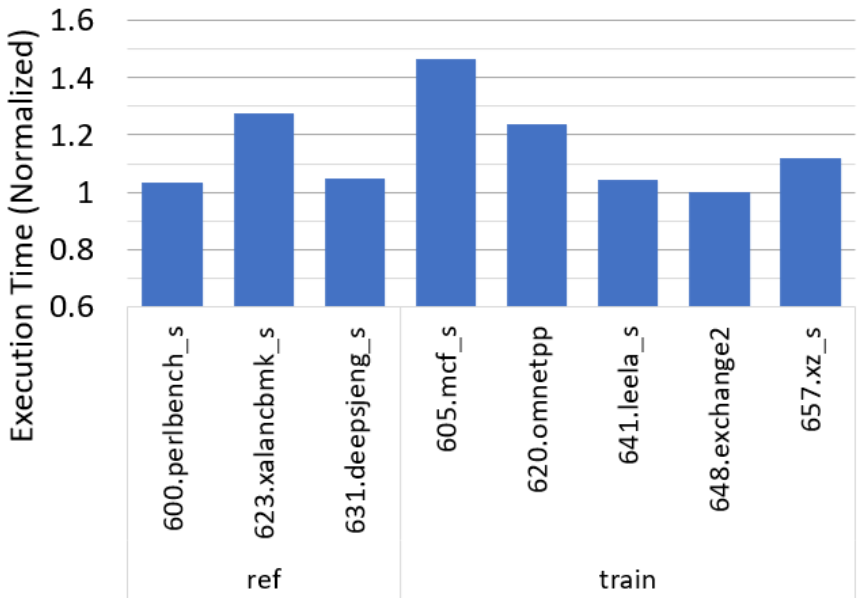


Figure 7.1: Execution time on the ReRAM system normalized to the DRAM only system.

of the fastest clock on the target system. The ReRAM system's execution time normalized to the DRAM only system's execution time is shown in Figure 7.1. The increase in execution time ranges from virtually non-existent, such as in the case of 648.exchange2_s, to 47% in the case of 605.mcf_s.

A variety of factors influence how impactful a ReRAM monolithic memory system will be on the execution time. An obvious factor is how memory intensive the benchmark is. The MPKI for each benchmark is reported in Figure 7.2. The MPKI for 605.mcf_s is the highest at 9.8, which matches it having the largest increase in execution time. Benchmarks with MPKIs below 0.1 experience very little increase in execution time; 641.leela_s has an MPKI of 0.08 and its execution time increased by 4.2%.

MPKI does not explain all variations though. One factor we have previously found important is the read-write ratio. The breakdown of reads and writes at

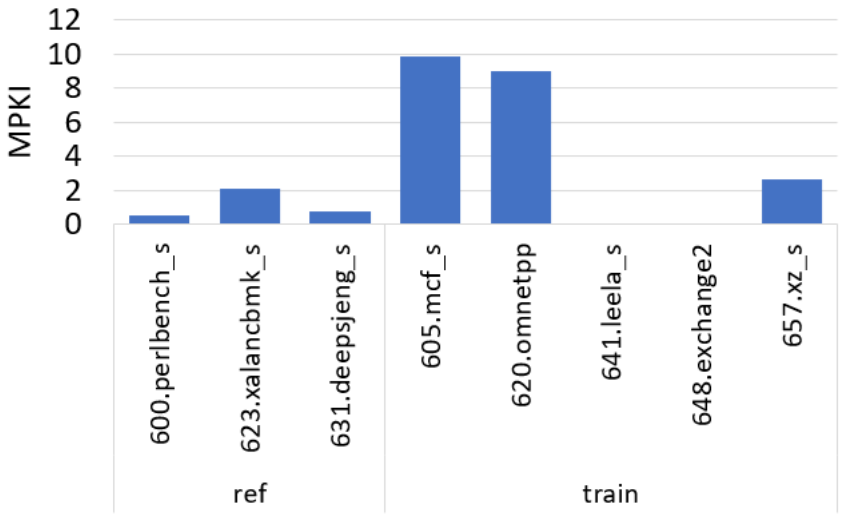


Figure 7.2: MPKI on the ReRAM system.

the main memory is shown in Figure 7.3. Since writes are more expensive than reads in terms of latency, and the ReRAM controller has no mechanism to prioritize reads ahead of writes, a large share of writes can negatively affect the execution time. For instance, 641.leela_s has a lower MPKI than 600.perlbench_s but has more than double the amount of write traffic, potentially contributing to 600.perlbench_s having a slowdown of 3.2% to 641.leela_s's 4.2%.

Another factor that impacts the performance of the ReRAM memory system is the memory traffic pattern. If there is significant queuing in the memory system, the ReRAM memory will do increasingly worse compared to a lower latency memory. This is because of the relatively high latency of ReRAM memory stacking in the queue. If there are 4 read requests queued at a ReRAM bank, the first request will have a latency of 200 ns, the second will have a latency of 400 ns, etc. This adds up to 2000 ns of main memory latency (800 ns read latency, 1200 ns queueing latency), compared to 150 ns (60 ns read latency, 90 ns queueing latency) if the memory

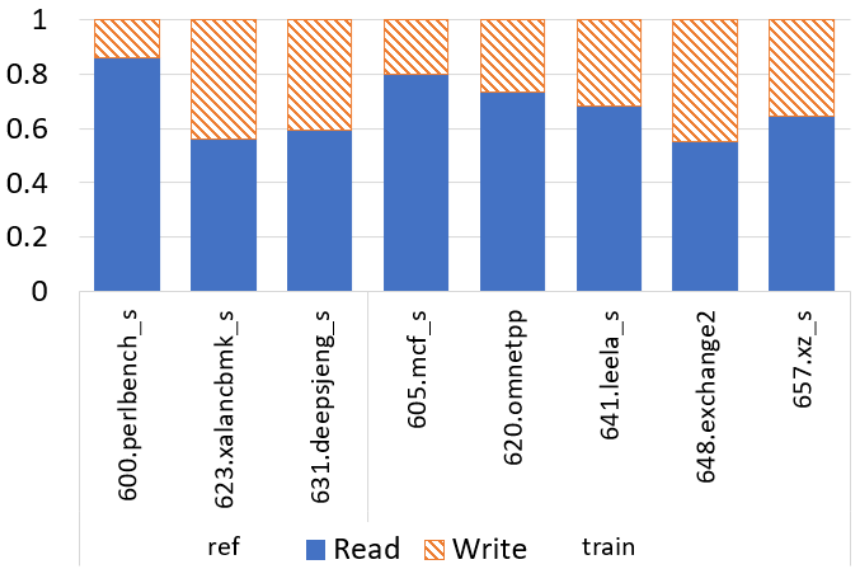


Figure 7.3: Read-write ratio.

latency is 15 ns; a 2.5x increase in the latency difference when compared to no queueing (740 ns vs 1850 ns). When we increase the number of queued requests to 8, it is a 4.5x increase in the latency difference. This makes the relative performance of the ReRAM memory system to the DRAM memory system particularly sensitive to queueing.

An increase in queueing, beyond that explained purely by MPKI, can be caused by systemic bank conflicts or by an uneven request rate across time. Based on Singh *et al.*'s memory analysis of the SPEC CPU2017 suite [120], 605.mcf_s has very a bursty memory traffic pattern, while 620.omntepp_s has relatively constant memory traffic. This means that while 620.omntepp_s has nearly the same MPKI as 605.mcf_s, its execution time only increases by 24% compared to 47% for 605.mcf_s. Similarly, 623.xalancbmk_s has an MPKI of 2.1, but has about the same slow down, 27%, as 620.omntepp_s because it frontloads the majority of its memory traffic.

Overall, the ReRAM memory system performed surprisingly well against the

DRAM memory system in these benchmarks, slowing down 15.3% on average. This may be an indication that it would be reasonable to use ReRAM memory in place of DRAM for lower performance devices.

7.4 Area

While we were able to use Cacti to analyze the additional area required when integrating ReRAM over an SRAM cache in Chapter 3, we were unable to examine the impact of combining the controllers. In this section, we look at the additional resources required to implement a combined cache-main memory controller.

For the base controller, we used the same SiFive Inclusive L2 Cache as in the performance evaluation. We modified all major stores (the directory, tags, and data arrays) to be dummy modules so the area would only capture the controller. For our cache-main memory controller implementation, we modified this controller to keep track of the ReRAM bank state and schedule requests based on bank availability in addition to the priorities established by the base controller. The ReRAM banks were implemented similarly to the other arrays as dummy modules.

We synthesized both controllers using Xilinx Vivado. The FPGA resources utilized for each design and the percentage the base cache controller is of the total Dual BOOM SoC are reported in Table 7.3. The first line of the table is the Look Up Tables (LUT) which are the primary way that FPGAs implement logic functions. This increased to account for the added complexity in scheduling required when taking into account ReRAM bank availability. The next line is the Look Up Table

Table 7.3: Controller FPGA Resources Used

	Cache Only (Total Design %)	Combined	Increase %
LUT	5461 (4.3%)	6062	11.0%
LUTRAM	476 (9.6%)	480	0.8%
FF	2680 (4.0%)	3073	14.7%

RAM, which is used when small blocks of RAM are required. The final line is Flip Flops (FF) which are the storage device used for unaddressed memory. Both of these increased with the increase in state needing to be maintained for each ReRAM bank.

Adding the memory controller functions to the L2 cache controller resulted in a moderate increase in resource utilization. Our ReRAM controller is relatively simple compared to many modern DRAM controllers; it does not re-order requests, try to maximize row buffer locality or need to perform refreshes. This makes it straightforwardly integrated into the L2 cache controller with a manageable increase in resources.

7.5 Conclusion

We have implemented a Dual BOOM SoC that includes a ReRAM memory model. Using this, we characterized the slowdown of the monolithic ReRAM memory system compared to a DDR3 memory system on SPECspeed 2017 Integer benchmarks. We found that there was an average slowdown of 15.3%. The actual slow down depends on the memory intensiveness, the read-write ratio, and the memory access pattern of each benchmark.

We additionally modeled our ReRAM-cache controller, and found it created up to a 14.7% increase in FPGA resource utilization. This increase comes from additional scheduling complexity and an increase in the amount of state the controller requires.

Finally, the tools used to create our SoC provide a path to manufacturing a test chip. The actual memory interface would need to be implemented, as well as the careful layout of the ReRAM and cache arrays, but the remaining components of the proposed system can be used to create an ASIC for testing our monolithic integration.

Chapter 8: Conclusion

Traditional main memories are running into scaling issues and the bandwidth wall due to pin limitations from the CPU to off-chip main memory. Emerging non-volatile memories and 3-D technologies provide opportunities for mitigating these problems. Crosspoint ReRAM is one such technology. In crosspoint ReRAM, the memory cells and a selector device are sandwiched between two metal wires. This allows many layers of memory, with unrelated circuits beneath, allowing a monolithically integrated CPU-main memory chip, increasing density and effective bandwidth.

This thesis shows the feasibility of placing crosspoint arrays over the last-level cache. Using Cacti to co-design the cache arrays and crosspoint arrays, we show with 8-layer ReRAM, we can integrate 1 GB of ReRAM over 2 MB of SRAM cache. We find that 84% of the cache can be covered by ReRAM with acceptable impacts to cache which reduces the total area of the ReRAM and cache by 30%.

A promising target architecture for this type of memory system is a manycore with multi-threading and wide SIMD. We propose to distribute the ReRAM such that every core is physically near a portion of the overall main memory. We also discuss the streamlining opportunities presented by integrating the main memory

into the last-level cache—namely, combining the controllers and communication interconnects. Finally, we develop a cache pipeline scheme that incorporates main memory accesses to increase the cache parallelism while maintaining the streamlined interface. Using area models from McPAT and our modified Cacti tool, we show that a 256 tile manycore is feasible on a KNL-sized die.

We develop a simulator that models this type of architecture with an accurate memory model which includes our streamlined cache–main memory interface. Using this simulator, we show the monolithic memory system outperforms state-of-the-art HBM2 DRAM for very memory intensive benchmarks. For the graph benchmarks, the monolithic ReRAM system improved performance by 5.3x and 2.7x for 4- and 8-stacks of HBM2 memory. In the case of streaming benchmarks, the monolithic ReRAM system has 1.4x the performance of the 4-stack HBM2 memory and is within 10% of the performance of the 8-stack HBM2 system.

We further developed an analytic model to describe the parallelism of our monolithic system. The model considers each of the subsystems and their overall relationship. Using this model, we explore the architectural balance of the systems we previously simulated, demonstrating where the base configuration was over- or under-provisioned. We found the compute side had 8x–12x more compute resources than the memory system could support for our most memory intensive benchmarks, and that the network size could reasonably be reduced by a factor of 4, but that the cache and main memory were nearly balanced with each other.

Finally, we created a RISC-V SoC with out-of-order cores to evaluate the impact of monolithic ReRAM main memory on more compute bound and latency

sensitive benchmarks. We found the performance to degrade by 15.3% on average. We additionally created an RTL model for the combined cache–main memory controller and demonstrated that it increases resource utilization by up to 14.7%. In creating a full RISC-V system, we've take the first step towards creating a test chip for our monolithic memory system.

Chapter 9: Further Work

9.1 System on a Chip

We have developed the RTL models for a dual core system which emulates a ReRAM memory system. We have run a single configuration on a limited number of benchmarks. It would be enlightening to try other configurations with varied workloads. It would also be beneficial to manufacture this system with actual ReRAM over the last-level cache to demonstrate the feasibility of a monolithically integrated main memory system and promote further research into its capabilities and limitations. The tools we used contain a path toward VLSI implementations and we have a basis for how to physically layout the integrated the ReRAM and cache.

9.2 Endurance and Reliability

We assume that acceptable endurance for monolithically integrated ReRAM main memory can be achieved through wear-leveling. Wear-leveling research has often focused on block memory schemes such as those found in flash. These schemes rely on tables to keep track of writes to each chunk of memory. The overhead of such schemes becomes untenable when working with byte-addressable memories. A few

schemes exist for a more fine-grained approach to wear leveling, but further research into wear-leveling techniques for main memories using lower endurance non-volatile memories is an important avenue of future research as more NVMs are used as main memory rather than storage.

Additionally, circuit level research into increasing the endurance of the memory cells could be a different avenue in solving this issue for monolithic systems. Our memory scheme did not rely heavily on the non-volatile aspect of the ReRAM memory. If different write energies/durations could trade-off endurance for a reduction in memory retention as suggested by Jagasivamani [79], the optimal balance between the two is an interesting research question.

9.3 Alternate Architectures

This thesis focuses on a single target architecture—a manycore CPU. Future work should look at monolithic ReRAM within other architectures. A natural fit would be GPUs which focus on throughput computation natively. GPUs generally do not have as much cache as CPUs, so an important consideration is what part of the GPU would be well suited to integrate the ReRAM over, if any. Another possibility is lower performance, integrated devices where the density and non-volatility of crosspoint ReRAM would be greatly beneficial and slower performance is less of a concern. Machine learning inference tasks on low performance embedded devices might be made possible with the large capacities with low static power offered by monolithically integrated ReRAM.

Bibliography

[1] Anant Agarwal. Limits on interconnection network performance. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):398–412, 1991.

[2] Chun Jason Xue, Youtao Zhang, Yiran Chen, Guangyu Sun, J. Jianhua Yang, and Hai Li. Emerging non-volatile memories: Opportunities and challenges. In *Proceedings of the Seventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '11, page 325–334, New York, NY, USA, 2011. Association for Computing Machinery.

[3] Dmytro Apalkov, Alexey Khvalkovskiy, Steven Watts, Vladimir Nikitin, Xueti Tang, Daniel Lottis, Kiseok Moon, Xiao Luo, Eugene Chen, Adrian Ong, Alexander Driskill-Smith, and Mohamad Krounbi. Spin-transfer torque magnetic random access memory (stt-mram). *J. Emerg. Technol. Comput. Syst.*, 9(2), May 2013.

[4] Cong Xu, Dimin Niu, Naveen Muralimanohar, Rajeev Balasubramonian, Tao Zhang, Shimeng Yu, and Yuan Xie. Overcoming the Challenges of Crossbar Resistive Memory Architectures. In *Proceedings of International Symposium on High Performance Computer Architecture*, 2015.

[5] Brian M. Rogers, Anil Krishna, Gordon B. Bell, Ken Vu, Xiaowei Jiang, and Yan Solihin. Scaling the bandwidth wall: Challenges in and avenues for cmp scaling. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, page 371–382, New York, NY, USA, 2009. Association for Computing Machinery.

[6] Crossbar. ReRAM Memory, Crossbar. https://www.crossbar-inc.com/assets/resources/white-papers/Crossbar-ReRAM-Technology.pdf 2017.

[7] Intel. Intel Optane Technology. http://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html 2017.

[8] Meenatchi Jagasivamani, Candace Walden, Devesh Singh, Luyi Kang, Shang Li, Mehdi Asnaashari, Sylvain Dubois, Bruce Jacob, and Donald Yeung. Analyzing the Monolithic Integration of a ReRAM-based Main Memory into a

CPU's Die. *IEEE Micro, Special Issue on Monolithic 3D Architectures*, 39(6), November/December 2019.

[9] Ady Tal. Intel software development emulator. `https://software.intel.com/content/www/us/en/develop/articles/intel-software-development-emulator.html`, 2020.

[10] Ishwar Bhati, Mu-Tien Chang, Zeshan Chishti, Shih-Lien Lu, and Bruce Jacob. Dram refresh mechanisms, penalties, and trade-offs. *IEEE Transactions on Computers*, 65(1):108–121, 2016.

[11] Uksong Kang, Hak-Soo Yu, Churoo Park, Hongzhong Zheng, John Halbert, Kuljit Bains, S Jang, and Joo Sun Choi. Co-architecting controllers and dram to enhance dram process scaling. In *The memory forum*, volume 14, 2014.

[12] Sung-Kye Park. Technology scaling challenge and future prospects of dram and nand flash memory. In *2015 IEEE International Memory Workshop (IMW)*, pages 1–4, 2015.

[13] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, and Joel Emer. High performance cache replacement using re-reference interval prediction (rrip). In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, page 60–71, New York, NY, USA, 2010. Association for Computing Machinery.

[14] Armin Vakil-Ghahani, Sara Mahdizadeh-Shahri, Mohammad-Reza Lotfi-Namin, Mohammad Bakhshalipour, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. Cache replacement policy based on expected hit count. *IEEE Computer Architecture Letters*, 17(1):64–67, 2018.

[15] Xi Chen, Lei Yang, Robert P. Dick, Li Shang, and Haris Lekatsas. C-pack: A high-performance microprocessor cache compression algorithm. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 18(8):1196–1208, 2010.

[16] An Chen. A review of emerging non-volatile memory (nvm) technologies and applications. *Solid-State Electronics*, 125:25–38, 2016. Extended papers selected from ESSDERC 2015.

[17] Geoffrey W. Burr, Matthew J. Breitwisch, Michele Franceschini, Davide Garetto, Kailash Gopalakrishnan, Bryan Jackson, Bülent Kurdi, Chung Lam, Luis A. Lastras, Alvaro Padilla, Bipin Rajendran, Simone Raoux, and Rohit S. Shenoy. Phase change memory technology. *Journal of Vacuum Science & Technology B*, 28(2):223–262, 2010.

[18] Scott W. Fong, Christopher M. Neumann, and H.-S. Philip Wong. Phase-change memory—towards a storage-class memory. *IEEE Transactions on Electron Devices*, 64(11):4374–4385, 2017.

[19] Xiangyu Dong, Cong Xu, Yuan Xie, and Norman P. Jouppi. NVSim: A Circuit-level Performance, Energy, and Area Model for Emerging Nonvolatile Memory. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 31(7), July 2012.

[20] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. A durable and energy efficient main memory using phase change memory technology. volume 37, pages 14–23, 06 2009.

[21] Satoru Hanzawa, Naoki Kitai, Kenichi Osada, Akira Kotabe, Yuichi Matsui, Nozomu Matsuzaki, Norikatsu Takaura, Masahiro Moniwa, and Takayuki Kawahara. A 512kb embedded phase change memory with 416kb/s write throughput at 100ua cell write current. In *2007 IEEE International Solid-State Circuits Conference. Digest of Technical Papers*, pages 474–616, 2007.

[22] Benjamin Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable dram alternative. volume 37, pages 2–13, 06 2009.

[23] Geoffrey W. Burr, Matthew J. Brightsky, Abu Sebastian, Huai-Yu Cheng, Jau-Yi Wu, Sangbum Kim, Norma E. Sosa, Nikolaos Papandreou, Hsiang-Lan Lung, Haralampos Pozidis, Evangelos Eleftheriou, and Chung H. Lam. Recent progress in phase-change memory technology. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 6(2):146–162, 2016.

[24] Rajendra Bishnoi, Mojtaba Ebrahimi, Fabian Oboril, and Mehdi B. Tahoori. Improving write performance for stt-mram. *IEEE Transactions on Magnetics*, 52(8):1–11, 2016.

[25] Sang Phill Park, Sumeet Gupta, Niladri Mojumder, Anand Raghunathan, and Kaushik Roy. Future cache design using stt mrams for improved energy efficiency: Devices, circuits and architecture. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, page 492–497, New York, NY, USA, 2012. Association for Computing Machinery.

[26] Mohamed M. Sabry Aly, Tony F. Wu, Andrew Bartolo, Yash H. Malviya, William Hwang, Gage Hills, Igor Markov, Mary Wootters, Max M. Shulaker, H.-S. Philip Wong, and Subhasish Mitra. The N3XT Approach to Energy-Efficient Abundant-Data Computing. *Proceedings of the IEEE*, 107(1), January 2019.

[27] Jimmy J. Kan, Chando Park, Chi Ching, Jaesoo Ahn, Yuan Xie, Mahendra Pakala, and Seung H. Kang. A study on practically unlimited endurance of stt-mram. *IEEE Transactions on Electron Devices*, 64(9):3639–3646, 2017.

[28] Lunkay Zhang, Brian Neely, Diana Franklin, Dmitri Strukov, Yuan Xie, and Frederic T. Chong. Mellow Writes: Extending Lifetime in Resistive Memories through Selective Slow Write Backs. In *Proceedings of the 43rd International Symposium on Computer Architecture*, 2016.

[29] Subramanya R. Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. Data Tiering in Heterogeneous Memory Systems. In *Proc. 11th European Conf. on Computer Systems*, 2016.

[30] Dimin Niu, Cong Xu, Naveen Muralimanohar, Norman P. Jouppi, and Yuan Xie. Design of cross-point metal-oxide reram emphasizing reliability and cost. In *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 17–23, 2013.

[31] Myoung-Jae Lee, Chang Bum Lee, Dongsoo Lee, Seung Ryul Lee, Man Chang, Ji Hyun Hur, Young-Bae Kim, Chang-Jung Kim, David H. Seo, Sunae Seo, U-In Chung, In-Kyeong Yoo, and Kinam Kim. A Fast, High-Endurance and Scalable Non-Volatile Memory Device Made from Asymmetric Ta2O5-x/TaO2-x Bilayer Structures. *Nature Materials*, 10, 2011.

[32] Meenatchi Jagasivamani, Candace Walden, Devesh Singh, Luyi Kang, Mehdi Asnaashari, Sylvain Dubois, Bruce Jacob, and Donald Yeung. Tileable Monolithic ReRAM Memory Design. In *Proceedings of the IEEE Symposium on Low-Power and High-Speed Chips and Systems*, Tokyo, Japan, April 2020.

[33] Shouhei Fukuyama, Atsuna Hayakawa, Ryutaro Yasuhara, Shinpei Matsuda, Hiroshi Kinoshita, and Ken Takeuchi. Comprehensive analysis of data-retention and endurance trade-off of 40nm tao¡inf¿x¡/inf¿-based reram. In *2019 IEEE International Reliability Physics Symposium (IRPS)*, pages 1–6, 2019.

[34] Xiaoxia Wu, Jian Li, Lixin Zhang, Evan Speight, Ram Rajamony, and Yuan Xie. Hybrid cache architecture with disparate memory technologies. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, page 34–45, New York, NY, USA, 2009. Association for Computing Machinery.

[35] Moinuddin K. Qureshi, Vijayalakshmi, and Jude A. Rivers. Scalable High Performance Main Memory System Using Phase-Change Memory Technology. In *Proc. International Symposium on Computer Architecture*, 2009.

[36] Dmitrii Ustiugov, Alexandros Daglis, Javier Picorel, Mark Sutherland, Edouard Bugnion, Babak Falsafi, and Dionisios Pnevmatikatos. Design Guidelines for High-Performance SCM Hierarchies. In *Proceedings of the 4th International Symposium on Memory Systems*, National Harbor, DC, October 2018.

[37] Gaurav Dhiman, Raid Ayoub, and Tajana Rosing. PDRAM: A Hybrid PRAM and DRAM Main Memory System. In *Proceedings of the Design Automation Conference*, San Francisco, CA, July 2009.

[38] Soyoon Lee, Hyokyung Bahn, and Sam H. Noh. CLOCK-DWF: A Write-History-Aware Page Replacement Algorithm for Hybrid PCM and DRAM Memory Architectures. *IEEE Transactions on Computers*, 63(9), 2014.

[39] Neha Agarwal and Thomas F. Wenisch. Thermostat: Application-transparent Page Management for Two-tiered Main Memory. In *Proceedings of the International Symposium on Architectural Support for Programming Languages and Operating Systems*, Xi'an, China, April 2017.

[40] Luiz Ramos, Eugene Gorbatov, and Ricardo Bianchini. Page Placement in Hybrid Memory Systems. In *Proc. 2011 International Conf. on Supercomputing*, 2011.

[41] Doris Keitel-Schulz and Norbert Wehn. Embedded DRAM Development: Technology, Physical Design, and Application Issues. *IEEE Design & Test of Computers*, May-June 2001.

[42] Bruce Jacob, Spencer Ng, and David T. Wang. *DRAM- vs Logic-Optimized Process Technology*, page 375–376. Morgan Kaufmann Publishers, 2010.

[43] Gabriel H. Loh. 3d-stacked memory architectures for multi-core processors. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, page 453–464, USA, 2008. IEEE Computer Society.

[44] Ronald G. Dreslinski, David Fick, Bharan Giridhar, Gyouho Kim, Sangwon Seo, Matthew Fojtik, Sudhir Satpathy, Yoonmyung Lee, Daeyeon Kim, Nurrachman Liu, Michael Wieckowski, Gregory Chen, Trevor Mudge, Dennis Sylvester, and David Blaauw. Centip3de: A 64-core, 3d stacked, near-threshold system. pages 1–30, 2012.

[45] Hongshin Jun, Jinhee Cho, Kangseol Lee, Ho-Young Son, Kwiwook Kim, Hanho Jin, and Keith Kim. Hbm (high bandwidth memory) dram technology and architecture. In *2017 IEEE International Memory Workshop (IMW)*, pages 1–4, 2017.

[46] Matt Poremba, Sparsh Mittal, Dong Li, Jeffrey S. Vetter, and Yuan Xie. Destiny: A tool for modeling emerging 3d nvm and edram caches. In *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1543–1546, 2015.

[47] Yuh-Fang Tsai, Yuan Xie, N. Vijaykrishnan, and Mary Jane Irwin. Three-dimensional cache design exploration using 3dcacti. In *Proceedings of the 2005 International Conference on Computer Design*, ICCD '05, page 519–524, USA, 2005. IEEE Computer Society.

[48] Dae Hyun Kim, Krit Athikulwongse, Michael B. Healy, Mohammad M. Hossain, Moongon Jung, Ilya Khorosh, Gokul Kumar, Young-Joon Lee, Dean L. Lewis, Tzu-Wei Lin, Chang Liu, Shreepad Panth, Mohit Pathak, Minzhen

Ren, Guanhao Shen, Taigon Song, Dong Hyuk Woo, Xin Zhao, Joungho Kim, Ho Choi, Gabriel H. Loh, Hsien-Hsin S. Lee, and Sung Kyu Lim. Design and analysis of 3d-maps (3d massively parallel processor with stacked memory). *IEEE Transactions on Computers*, 64(1):112–125, 2015.

[49] Rakesh Anigundi, Hongbin Sun, Jian-Qiang Lu, Ken Rose, and Tong Zhang. Architecture design exploration of three-dimensional (3d) integrated dram. In *2009 10th International Symposium on Quality Electronic Design*, pages 86–90, 2009.

[50] Yuhua Guo, Weijun Xiao, Qing Liu, and Xubin He. A cost-effective and energy-efficient architecture for die-stacked dram/nvm memory systems. In *2018 IEEE 37th International Performance Computing and Communications Conference (IPCCC)*, pages 1–2, 2018.

[51] Wangyuan Zhang and Tao Li. Exploring Phase Change Memory and 3D Die-Stacking for Power/Thermal Friendly, Fast and Durable Memory Architectures. In *Proceedings of the International Symposium on Parallel Architectures and Compilation Techniques*, 2009.

[52] Guangyu Sun, Xiangyu Dong, Yuan Xie, Jian Li, and Yiran Chen. A novel architecture of the 3d stacked mram l2 cache for cmps. In *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, pages 239–249, 2009.

[53] Nauman H. Khan, Syed M. Alam, and Soha Hassoun. Power delivery design for 3-d ics using different through-silicon via (tsv) technologies. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 19(4):647–658, 2011.

[54] Shadi M.S. Harb and William Eisenstadt. Thermal evaluation of tsvs in 3d-integration technology. In *2019 IEEE 21st Electronics Packaging Technology Conference (EPTC)*, pages 1–4, 2019.

[55] Abdul Hamid Bin Yousuf, Nahid M. Hossain, and Masud H. Chowdhury. Impacts of different shapes of through-silicon-via core on 3d ic performance. In *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–4, 2017.

[56] Mindy D. Bishop, H.-S. Philip Wong, Subhasish Mitra, and Max M. Shulaker. Monolithic 3-d integration. *IEEE Micro*, 39(6):16–27, 2019.

[57] Mohamed M. Sabry Aly, Mingyu Gao, Gage Hills, Chi-Shuen Lee, Greg Pitner, Max M. Shulaker, Tony F. Wu, Mehdi Asheghi, Jeff Bokor, Franz Franchetti, Kenneth E. Goodson, Christos Kozyrakis, Igor Markov, Kunle Olukotun, Larry Pileggi, Eric Pop, Jan Rabaey, Christopher Re, H.-S. Philip Wong, and Subhasish Mitra. Energy-Efficient Abundant-Data Computing: The N3XT 1,000x. *Computer*, 2015.

[58] Max M. Shulaker, Gage Hills, Rebecca S. Park, Roger T. Howe, Krishna Saraswat, H.-S. Philip Wong, and Subhasish Mitra. Three-dimensional integration of nanotechnologies for computing and data storage on a single chip. *Nature*, 547:74–78, 2017.

[59] Kyungwook Chang, Sai Pentapati, Da Eun Shim, and Sung Kyu Lim. Road to high-performance 3d ics: Performance optimization methodologies for monolithic 3d ics. In *Proceedings of the International Symposium on Low Power Electronics and Design*, ISLPED '18, New York, NY, USA, 2018. Association for Computing Machinery.

[60] Sai Pentapati, Lingjun Zhu, Lennart Bamberg, Da Eun Shim, Alberto García-Ortiz, and Sung Kyu Lim. A logic-on-memory processor-system design with monolithic 3-d technology. *IEEE Micro*, 39(6):38–45, 2019.

[61] David Manners. Darpa 3dsoc cnfet project moves towards commercialisation phase. Aug 2020.

[62] Sung Hyun Jo, Kuk-Hwan Kim, and Wei Lu. High-Density Cross-bar Arrays Based on a Si Memristive System. *Nano Letters*, 2009.

[63] Sung Hyun Jo, T. Kumar, S. Narayanan, W. D. Lu, and H. Nazarian. 3D-stackable crossbar resistive memory based on Field Assisted Superlinear Threshold (FAST) selector. *IEEE International Elec-tron Devices Meeting*, 2014.

[64] Crossbar. Personal communication. 2020.

[65] Meenatchi Jagasivamani, Candace Walden, Devesh Singh, Luyi Kang, Shang Li, Mehdi Asnaashari, Sylvain Dubois, Donald Yeung, and Bruce Jacob. Design for ReRAM-based Main-Memory Architectures. In *Proceedings of the 5th International Symposium on Memory Systems*, Washington, D.C., September 2019.

[66] Moinuddin K. Qureshi, John Karidis, Michele Franceschini, Vijayalakshmi Srinivasan, Luis Lastras, and Bulent Abali. Enhancing Lifetime and Security of PCM-based Main Memory with Start-Gap Wear Leveling. In *Proceedings of the 42nd Annual International Symposium on Microarchitecture*, New York, NY, December 2009.

[67] Shyamkumar Thoziyoor, Naveen Muralimanohar, Jung Ho Ahn, and Norman Jouppi. Cacti 5.1. Technical report, HP Laboratories, April 2008.

[68] Naveen Muralimanohar, Rajeev Balasubramonian, and Norm Jouppi. Optimizing nuca organizations and wiring alternatives for large caches with cacti 6.0. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 3–14, Washington, DC, USA, 2007. IEEE Computer Society.

[69] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: A Many-Core x86 Architecture for Visual Computing. *ACM Transactions on Graphics*, 27(3), August 2008.

[70] http://www.theinquirer.net/?article=19423. Sun's Niagara Falls Neatly into Multithreaded Place. November 2004.

[71] Intel. AVX 512 Instruction Extensions, http://software.intel.com/en-us/blogs/2013/avx-512-instructions. 2017.

[72] John H. Kelm, Daniel R. Johnson, Matthew R. Johnson, Neal C. Crago, William Tuohy, Aqeel Mahesri, Steven S. Lumetta, Matthew I. Frank, and Sanjay J. Patel. Rigel: An Architecture and Scalable Programming Interface for a 1000-core Accelerator. In *Proceedings of the International Symposium on Computer Architecture*, pages 140–151, Austin, TX, June 2009.

[73] Tianhao Zheng, Haishan Zhu, and Mattan Erez. Sipt: Speculatively indexed, physically tagged caches. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 118–130, 2018.

[74] Amit Agarwal, Kaushik Roy, and T. N. Vijaykumar. Exploring high bandwidth pipelined cache architecture for scaled technology. In *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1*, DATE '03, pages 10778–, Washington, DC, USA, 2003. IEEE Computer Society.

[75] James Tuck, Luis Ceze, and Josep Torrellas. Scalable cache miss handling for high memory-level parallelism. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pages 409–422, Dec 2006.

[76] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *Proceedings of the International Symposium on Microarchitecture*, New York, NY, December 2009.

[77] Ian Cutress. Supercomputing 15: Intel's knights landing / xeon phi silicon on display. Nov 2015.

[78] Albert Lee, Chieh-Pu Lo, Chien-Chen Lin, Wei-Hao Chen, Kuo-Hsiang Hsu, Zhibo Wang, Fang Su, Zhe Yuan, Qi Wei, Ya-Chin King, Chrong-Jung Lin, Hochul Lee, Pedram Khalili Amiri, Kang-Lung Wang, Yu Wang, Huazhong Yang, Yongpan Liu, and Meng-Fan Chang. A reram-based nonvolatile flip-flop with self-write-termination scheme for frequent-off fast-wake-up nonvolatile processors. *IEEE Journal of Solid-State Circuits*, 52(8):2194–2207, 2017.

[79] Meenatchi Jagasivamani. *Resistive RAM Based Main-Memory Systems: Understanding The Opportunities, Limitations, And Tradeoffs*. PhD thesis, University of Maryland, College Park, 2020.

[80] Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2011.

[81] Trevor E. Carlson, Wim Heirman, Stijn Eyerman, Ibrahim Hur, and Lieven Eeckhout. An Evaluation of High-Level Mechanistic Core Models. *ACM Transactions on Architecture and Code Optimization*, April 2014.

[82] Jason E. Miller, Harshad Kasture, George Kurian, Charles Gruenwald III, Nathan Beckmann, Christopher Celio, Jonathan Eastep, and Anant Agarwal. Graphite: A Distributed Parallel Simulator for Multicores. In *Proceedings of the 16th International Symposium on High-Performance Computer Architecture*, January 2010.

[83] Daniel Sanchez and Christos Kozyrakis. ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems. In *Proceedings of the 40th International Symposium on Computer Architecture*, Tel-Aviv, Israel, June 2013.

[84] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005.

[85] Shang Li, Zhiyuan Yang, Dhiraj Reddy, Ankur Srivastava, and Bruce Jacob. DRAMsim3: A Cycle-Accurate, Thermal Capable Memory System Simulator. *IEEE Computer Architecture Letters*, 2019.

[86] Masab Ahmad, Farrukh Jijaz, Qingchuan Shi, and Omer Khan. CRONO: A Benchmark Suite for Multithreaded Graph Algorithms Executing on Futuristic Multicores. In *Proceedings of the 2015 IEEE International Symposium on Workload Characterization*, Atlanta, GA, October 2015.

[87] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization*, Austin, TX, October 2009.

[88] David A. Bader, John Feo, John Gilbert, Jeremy Kepner, David Koester, Eugene Loh, Kamesh Madduri, Bill Mann, and Theresa Meuse. HPCS Scalable Synthetic Compact Applications #2 Graph Analysis. August 2006.

[89] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-MAT: A Recursive Model for Graph Minig. In *Proceedings of the 2004 SIAM International Conference on Data Mining*, 2004.

[90] Saeed Maleki, Yaoqing Gao, Maria J. Garzar´n, Tommy Wong, and David A. Padua. An evaluation of vectorizing compilers. In *2011 International Conference on Parallel Architectures and Compilation Techniques*, pages 372–382, 2011.

[91] Yulei Sui, XIaokang Fan, Hao Zhou, and Jingling Xue. Loop-oriented array- and field-sensitive pointer analysis for automatic simd vectorization. In *Proceedings of the 17th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, Tools, and Theory for Embedded Systems*, LCTES 2016, page 41–51, New York, NY, USA, 2016. Association for Computing Machinery.

[92] Vasileios Porpodas, Rodrigo C. O. Rocha, and Luís F. W. Góes. Look-ahead slp: Auto-vectorization in the presence of commutative operations. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, CGO 2018, page 163–174, New York, NY, USA, 2018. Association for Computing Machinery.

[93] Vasileios Porpodas, Rodrigo C. O. Rocha, and Luís F. W. Góes. Vw-slp: Auto-vectorization with adaptive vector width. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, PACT '18, New York, NY, USA, 2018. Association for Computing Machinery.

[94] Mike O'Connor, Niladrish Chatterjee, Donghyuk Lee, John Wilson, Aditya Agrawal, and Stephen W. Keckler William J. Dally. Fine-Grained DRAM: Energy-Efficient DRAM for Extreme Bandwidth Systems. In *Proceedings of the 50th International Symposium on Microarchitecture*, Cambridge, MA, October 2017.

[95] Arun F. Rodrigues, Karl Scott Hemmert, Brian W. Barrett, Chad D Kersey, Ron A. Oldfield, M. Weston, R. Risen, Jonathan Cook, Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. The structural simulation toolkit. *SIGMETRICS Perform. Eval. Rev.*, 38(4):37–42, March 2011.

[96] H. T. Kung. Memory requirements for balanced computer architectures. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, ISCA '86, page 49–54, Washington, DC, USA, 1986. IEEE Computer Society Press.

[97] Steven Fortune and James Wyllie. Parallelism in random access machines. In *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, STOC '78, page 114–118, New York, NY, USA, 1978. Association for Computing Machinery.

[98] Alok Aggarwal, Ashok K. Chandra, and Marc Snir. Hierarchical memory with block transfer. In *Proceedings of the 28th Annual Symposium on Foundations of Computer Science*, SFCS '87, page 204–216, USA, 1987. IEEE Computer Society.

[99] Alok Aggarwal, Bowen Alpern, Ashok K. Chandra, and Marc Snir. A model for hierarchical memory. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, STOC '87, page 305–314, New York, NY, USA, 1987. Association for Computing Machinery.

[100] B. Alpern, L. Carter, E. Feig, and T. Selker. The uniform memory hierarchy model of computation. *Algorithmica*, 12(72), 1994.

[101] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. Logp: Towards a realistic model of parallel computation. *SIGPLAN Not.*, 28(7):1–12, July 1993.

[102] Wilfried Oed and O. Lange. On the effective bandwidth of interleaved memories in vector processor systems. *IEEE Trans. Comput.*, 34(10):949–957, October 1985.

[103] Qing Yang and Tao Yang. A memory interference model for regularly patterned multiple stream vector accesses. *IEEE Transactions on Parallel and Distributed Systems*, 6(5):520–530, 1995.

[104] S. Bardhan and D. A. Menascé. Predicting the effect of memory contention in multi-core computers using analytic performance models. *IEEE Transactions on Computers*, 64(8):2279–2292, 2015.

[105] B. M. Tudor, Y. M. Teo, and S. See. Understanding off-chip memory contention of parallel programs in multicore systems. In *2011 International Conference on Parallel Processing*, pages 602–611, 2011.

[106] L. Ma, R. D. Chamberlain, and K. Agrawal. Performance modeling for highly-threaded many-core gpus. In *2014 IEEE 25th International Conference on Application-Specific Systems, Architectures and Processors*, pages 84–91, 2014.

[107] Lin Ma, Kunal Agrawal, and Roger D. Chamberlain. A memory access model for highly-threaded many-core architectures. *Future Generation Computer Systems*, 30:202 – 215, 2014. Special Issue on Extreme Scale Parallel Architectures and Systems, Cryptography in Cloud Computing and Recent Advances in Parallel and Distributed Systems, ICPADS 2012 Selected Papers.

[108] L. Ma and R. D. Chamberlain. A performance model for memory bandwidth constrained applications on graphics engines. In *2012 IEEE 23rd International Conference on Application-Specific Systems, Architectures and Processors*, pages 24–31, 2012.

[109] Sunpyo Hong and Hyesoon Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. *SIGARCH Comput. Archit. News*, 37(3):152–163, June 2009.

[110] A. Li, S. L. Song, E. Brugel, A. Kumar, D. Chavarría-Miranda, and H. Corporaal. X: A comprehensive analytic model for parallel machines. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 242–252, 2016.

[111] SPEC. SPEC's Benchmarks. `https://www.spec.org/benchmarks.html`, 2017.

[112] RISC-V International. About RISC-V. `https://riscv.org/about`, 2020.

[113] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. The rocket chip generator. Technical Report UCB/EECS-2016-17, EECS Department, University of California, Berkeley, Apr 2016.

[114] G. Gibeling, Andrew L. Schultz, and K. Asanović. The ramp architecture & description language. 2006.

[115] The Regents of the University of California. Future Work - The Vector ("V") ISA Extension. `https://docs.boom-core.org/en/latest/sections/future-work.html#the-vector-v-isa-extension`, 2021.

[116] SiFive. Block inclusive cache sifive. `https://github.com/sifive/block-inclusivecache-sifive`, 2020.

[117] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, Qijing Huang, Kyle Kovacs, Borivoje Nikolic, Randy Katz, Jonathan Bachrach, and Krste Asanović. FireSim: FPGA-accelerated cycle-exact scale-out system simulation in the public cloud. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, pages 29–42, Piscataway, NJ, USA, 2018. IEEE Press.

[118] David Biancolin, Sagar Karandikar, Donggyu Kim, Jack Koenig, Andrew Waterman, Jonathan Bachrach, and Krste Asanovic. Fased: Fpga-accelerated simulation and evaluation of dram. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '19, page 330–339, New York, NY, USA, 2019. Association for Computing Machinery.

[119] Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter Mattson, and John D. Owens. Memory access scheduling. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ISCA '00, page 128–138, New York, NY, USA, 2000. Association for Computing Machinery.

[120] Sarabjeet Singh and Manu Awasthi. Memory centric characterization and analysis of spec cpu2017 suite. *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, Apr 2019.